

Computer Science Semester Project

SWT Tutorial

Patrick Salamin

supervised by

Paul-Louis Meylan
Prof. Claude Petitpierre

at the

Laboratoire de téléinformatique, EPFL
April 18, 2005

Contents

1	Why SWT?	9
2	Building blocks of an SWT application	11
3	Environment set-up	13
4	Your first SWT application	15
5	Running SWT applications outside of Eclipse	19
6	SWT packages	21
7	Dialogs	23
7.1	MessageBox	26
7.2	ColorDialog	28
7.3	DirectoryDialog	30
7.4	FontDialog	32
7.5	FileDialog	34
7.6	PrintDialog	36
8	Widgets	39
8.1	Widget Events	40
8.1.1	How to use the listeners (SelectionListener)	41
8.1.2	KeyListener	42
8.1.3	MouseListener	44
	MouseListener	44
	MouseMoveListener	45
	MouseTrackListener	46
8.1.4	TextListener	46
	ModifyListener	46
	VerifyListener	46
8.1.5	FocusListeners and TraverseListener	47

	FocusListener	47
	TraverseListener	47
8.2	Useful widgets	49
8.2.1	Buttons	49
8.2.2	Slider, Scale and ProgressBar widgets	51
	Slider and Scale	51
	ProgressBar	53
8.2.3	Text widget	54
8.2.4	List widget	56
8.3	Composite widgets	58
8.3.1	Table widget	60
	TableColumn	61
	TableItem	61
8.3.2	Combo widget	65
8.3.3	Tree	67
8.3.4	TabFolder	70
8.3.5	CoolBar	72
8.4	Menu Widgets	76
8.4.1	Menu	76
	Simple Menu	76
	Advanced Menu	78
8.4.2	Pop-Up Menu	82
8.5	A summary of controls having items	85
9	Layouts	87
9.1	FillLayout	89
9.2	RowLayout	90
9.3	GridLayout	92
9.4	FormLayout	95
9.5	StackLayout	98
A	SWT editors	101
B	An interesting custom widget example	103

List of Figures

2.1	An SWT application from different perspectives	11
3.1	Possible error at the run-time	13
4.1	The simplest SWT application source code	16
4.2	Simplest SWT application resulting from the preceding code .	17
5.1	Eclipse logo	20
7.1	Dialog class hierarchy	23
7.2	Simple MessageBox code sample	24
7.3	Most important style bit constants for the Dialogs for the icons	25
7.4	MessageBox code sample	26
7.5	MessageBox resulting from the previous code	27
7.6	ColorDialog code sample	28
7.7	ColorDialog resulting from the previous code	29
7.8	DirectoryDialog code sample	30
7.9	DirectoryDialog resulting from the previous code	31
7.10	FontDialog code sample	32
7.11	FontDialog resulting from the previous code	33
7.12	FileDialog code sample	34
7.13	FileDialog resulting from the previous code	35
7.14	PrintDialog code sample (1/2)	36
7.15	PrintDialog code sample (2/2)	37
7.16	PrintDialog resulting from the preceding code	38
8.1	Widget class hierarchy	39
8.2	Event widgets summary	40
8.3	SelectionListener sample code	41
8.4	SelectionAdapter sample code	41
8.5	ButtonListener sample code	42
8.6	ButtonAdapter sample code	42

8.7	KeyListener code sample	43
8.8	MouseListener code sample	45
8.9	VerifyListener code sample	47
8.10	TraverseListener code sample	48
8.11	Control class hierarchy	49
8.12	Button code sample	50
8.13	Button resulting from the previous code	50
8.14	Useful button style bit constants	51
8.15	Scale and Slider code sample	52
8.16	Scale and Slider resulting from the previous code	53
8.17	ProgressBar code sample	53
8.18	ProgressBar resulting from the previous code	54
8.19	Text sample code	55
8.20	Text resulting from the previous code	55
8.21	List sample code	56
8.22	List resulting from the previous code	57
8.23	Composite can contain other composite classes	58
8.24	Composite code sample	59
8.25	Composite resulting from the previous code	60
8.26	Table code sample	60
8.27	TableColumn code sample	61
8.28	TableItem code sample	61
8.29	Table source code example 1	62
8.30	Table source code example 2	63
8.31	Table source code example 3	64
8.32	Table resulting from the previous codes	65
8.33	Simplest Combo code sample	65
8.34	Combo code sample	66
8.35	Combo resulting from the previous code	67
8.36	Tree code sample	67
8.37	TreeItem source code example 1	68
8.38	TreeItem source code example 2	68
8.39	TreeImage code sample	69
8.40	TreeSelectionListener code sample	69
8.41	TreeListener code sample	70
8.42	Tree resulting from the previous codes	70
8.43	TabFolder sample code	71
8.44	TabFolder resulting from the previous code	72
8.45	CoolBar code sample	72
8.46	CoolBar Button source code example 1	73
8.47	CoolBar SelectionListener code sample	73

LIST OF FIGURES

8.48	CoolBar Button source code example 2	73
8.49	ToolBar code sample	74
8.50	CoolBar Point code sample	74
8.51	CoolBar setWrapIndices method code sample	74
8.52	CoolBar resulting from the previous codes	75
8.53	Menu (simple) code source example 1	76
8.54	MenuItem (simple) source code example 1	76
8.55	Menu (simple) source code example 2	77
8.56	MenuItem (simple) source code example 2	77
8.57	Menu (simple) Listener code source example 1	77
8.58	Menu (simple) resulting from the previous codes	78
8.59	MenuItem (advanced) source code example 3	78
8.60	Menu (advanced) Listener code source example 2	79
8.61	MenuItem (advanced) code source example 4	79
8.62	MenuItem (advanced) code source example 5	79
8.63	MenuItem (advanced) code source example 6	80
8.64	Menu (advanced) Listener code source example 3	80
8.65	Menu (advanced) sub-action code source example 1	81
8.66	Menu (advanced) sub-action code source example 2	81
8.67	Menu (advanced) sub-action code source example 3	81
8.68	Menu (advanced) sub-action code source example 4	82
8.69	Menu (advanced) resulting from the previous codes	82
8.70	PopUp menu source code example 1	83
8.71	PopUp menu source code example 2	83
8.72	PopUp menu Composite code sample	84
8.73	PopUp menu setMenu()	84
8.74	PopUp menu resulting from the previous codes	84
8.75	Widget needing Item summary	85
9.1	SWT Layout manager hierarchy	87
9.2	FillLayout code sample	89
9.3	FillLayout resulting from window resizing	89
9.4	RowLayout code sample	90
9.5	RowLayout wrapping sample	90
9.6	RowLayout margins and spacing sample	90
9.7	RowLayoutData code sample	91
9.8	RowLayoutData sample	91
9.9	GridLayout code sample	92
9.10	GridLayout number of columns and spacings sample	92
9.11	GridLayout with column of equal width sample	93
9.12	GridData code sample	93

9.13	GridData horizontal spanning sample	93
9.14	GridData with the largest cell width sample	94
9.15	GridData with horizontal alignment sample	94
9.16	FormLayout code sample	95
9.17	FormData and FormAttachement sample	95
9.18	FormData code sample	96
9.19	FormData resulting from the previous code	97
9.20	StackLayout code example 1/2	98
9.21	StackLayout code example 2/2	99
9.22	StackLayout resulting from the previous code	99
B.1	custom ZoomedScrolledCanvas code sample	104
B.2	custom setComp() method code sample	104
B.3	custom setupScale() method code sample	105
B.4	custom setupZoom() method code sample	105
B.5	custom setImage() method code sample	106
B.6	custom SelectionListener code sample	107
B.7	custom last step code sample	107
B.8	custom application resulting from the previous codes	108
B.9	custom application used 1	108
B.10	custom application used 2	109

Chapter 1

Why SWT?

First of all, let us just talk about what is SWT ? It is a cross platform GUI developed by IBM. But why did IBM create another GUI library ? Why do not they have used the already existing Java GUI frameworks ? Let us return a little in the past of the Java history...

At first, Sun created a cross platform GUI framework AWT (Abstract Windowing Toolkit). This toolkit uses native widgets but it suffers from an LCD problem which causes loss of major platform features. More concretely, if a platform (P1) tolerate the widgets 1 to 40 and an other platform (P2) tolerate the widgets 20 to 25, then the cross-platform AWT framework only offers the intersection of these two sets.

In order to solve this problem, Sun created a new framework that uses emulated widgets instead of native widgets: Swing. This approach solved the LCD problem and provided a rich set of widgets but created other problems. For instance, Swing applications no longer look like the native applications. Although they are the latest improvements in the JVM, the Swing applications suffer performance problems that do not exist in their native counterparts. Moreover, the Swing applications consume too much memory, which is not suitable for small devices such as PDAs and Mobile Phones.

IBM decided that no one of these approaches fulfill their requirements. So, IBM created a new GUI library, called SWT (Standard Widget Toolkit), which solves the problems seen with the AWT and the Swing frameworks. The SWT framework accesses native widgets through JNI (Java Native Interface). If a widget is not available on the host platform, SWT emulates it.

Chapter 2

Building blocks of an SWT application

The *Display*, *Shell*, and *Widgets* elements are the basic building blocks in an SWT application. The *Display* instance is responsible from managing event loops and controlling communication between the UI thread and the other threads and the *Shell* instance corresponds to the window in an application managed by the OS window manager. In an SWT application one *Display* and one or more *Shell* instances are needed at least. The *Widget* instances are elements we will put on the window (we will describe them and their use in the chapter 8).

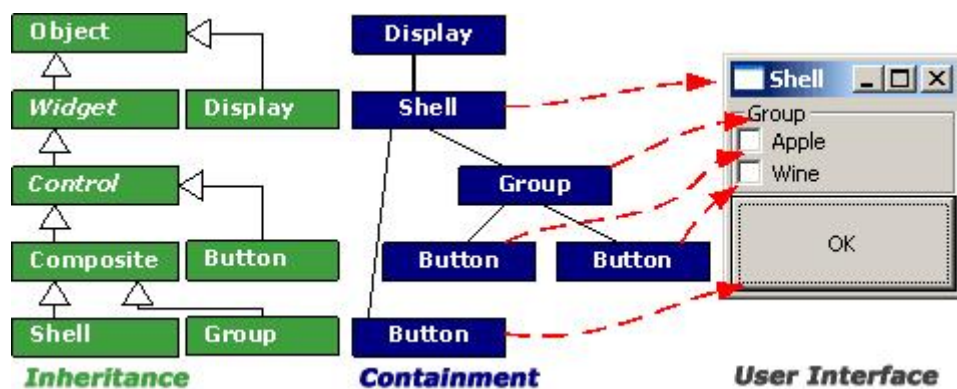


Figure 2.1: An SWT application from different perspectives

-
- The diagram on the left is the simplified inheritance diagram of the UI objects
 - The centered diagram is the containment structure of the UI objects
 - The diagram on the right is the created UI

Each thread of an application (they can be several) uses its own instance of a *Display* object. The current active instance of a *Display* object can be obtained by using the static `Display.getCurent()` method. The *Shell* object represents a window in a particular operating system. It can be normal, maximized or minimized. There are two types of *Shells*:

1. the top-level *Shell* created as a child, main window of the *Display*
2. the dialog *Shell* that depends on the other *Shells*

Shell type depends on the style bit constant(s) passed to the *Shell* constructor. If a *Display* object is given to the parameter, it is a top-level shell, otherwise the default value of a *Shell* is *DialogShell*.

Some widget properties must be set at the creation time. These widget properties are called style bits. They are defined as constants in the SWT class, e.g. `Button button = new Button(shell, <style bit(s)>)`. It is possible to use more than one style bit by using the OR operator (`|`). For instance, to use a bordered "push button", you need to use `SWT.PUSH | SWT.BORDER` in the style bit parameters.

Chapter 3

Environment set-up

To develop an SWT application is quite different from developing a Swing application. At first, you need to add the SWT libraries to your classpath and set the necessary environment variables.

The first library that you need is the *swt.jar* file that is under the `ECLIPSE_HOME\eclipse\plugins\org.eclipse.swt.win32_3.1.0\ws\win32` directory. Depending on the version of the Eclipse that you are currently using, you might need to use a different directory. The *swt.jar* file must be added to your classpath to make this go to menu *Project -> Properties -> JavaBuildPath -> Libraries -> Add Variable -> Eclipse Home -> Extend* and select the *swt.jar* library under the directory above and then click *OK*. Afterwards, you will be able to compile an SWT application but, because of a runtime exception shown below, you will not be able to execute it because the *swt.jar* library uses native libraries. So you need to set the `java.library.path` environment variable to use native libraries in Java.



```
Console output
java.lang.UnsatisfiedLinkError: no swt-win32-2133 in java.library.path
at java.lang.ClassLoader.loadLibrary(ClassLoader.java:1403)
at java.lang.Runtime.loadLibrary0(Runtime.java:788)
at java.lang.System.loadLibrary(System.java:832)
...
at org.eclipse.swt.widgets.Display.<init>(Display.java:287)
at Main.main(Main.java:25)
Exception in thread "main"
```

Figure 3.1: Possible error at the run-time

To set the `java.library.path` variable, go to *Run -> Run... -> Java Application -> New -> Arguments -> VM Arguments*. Thereafter, if necessary, modify the path below and paste it to the VM Arguments field.

`-Djava.library.path=ECLIPSE_HOME\eclipse\plugins\org.eclipse.swt.win32_3.1.0\os\win32\x86` *If you need to load any native library that your application uses, you can use the method:*
`Runtime.getPlatform.loadLibrary(LIBRARY_NAME)`

After these steps you will be able to run an SWT application within your eclipse environment.

Chapter 4

Your first SWT application

To Create a typical SWT application requires the following steps:

- Create a *Display* object
- Create one or more *Shell* objects
- Set a *Layout* manager of the Shell
- Create *Widget*(s) inside the *Shell*(s) (as many as you want)
- Open the *Shell* window
- Write an event dispatching loop
- Dispose the *Display* object

You can use the following code template to quickly run the code parts given later in this tutorial.

```
package ch.epfl.lti.basics;

import org.eclipse.swt.layout.RowLayout;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

public class FirstProgram {

    public FirstProgram() {
        Display display = new Display();    // 1

        Shell shell = new Shell(display);  // 2

        shell.setLayout(new RowLayout());  // 3

        shell.setText("My first program");
        shell.setSize(300, 200);            // 4

        /*****
         * Your code will come here *
         *****/

        shell.open();                      // 5

        while (!shell.isDisposed()) {
            if (!display.readAndDispatch())
                display.sleep();
        }                                  // 6

        display.dispose();                  // 7
    }

    public static void main(String[] args) {
        new FirstProgram();
    }
}
```

Figure 4.1: The simplest SWT application source code

This example displays an empty window in which you can add your widgets to the template above. As said before, every SWT application requires (1) a *Display* and one or more (2) *Shells*. The *Shell* is a *Composite* object: it can contain other *Composite* objects. If the (3) *Layout* of the *Shell* is not set, added *widgets* to the *Shell* will not be visible. You can (4) set the size and the title of your window. The *Shell* window must be (5) opened to be (6) displayed. The event handling loop reads and dispatches the GUI *Events*. If there is no event handling loop, the application window cannot be shown, even if the *Shell* window is opened by the `open()` method. Afterwards, you should dispose the (7) *Display*, when the *Shell* is discarded. The result of this code can be seen here:



Figure 4.2: Simplest SWT application resulting from the preceding code

You can use the Source -> Organize Imports menu or Ctrl+Shift+O to automatically import the required libraries.

Chapter 5

Running SWT applications outside of Eclipse

To run your application outside of Eclipse, the `swt.jar` library must be in your classpath, and the `java.library.path` environment variable must be set properly. Depending on the host platform, the appropriate native library file must be available:

1. Put the `swt.dll` file in the `c:\windows\system32` directory (or for the UNIX users, put the "so" file in the repository where your operating system usually goes to find the native libraries).
2. Put absolute path of your `swt.jar`
(`ECLIPSE_HOME\eclipse\plugins\org.eclipse.swt.win32.3.1.0\win32\swt.jar`)
in your CLASSPATH environment variable (try `WINDOWS_KEY (= the flag) + Pause -> Advanced -> Environment Variables -> CLASSPATH` and add the previous path)
(or for the UNIX users, set your CLASSPATH environment variable by adding the absolute path to your file `swt.jar` to it.)

Then you will only have to put the following lines in your terminal to compile and launch your application:

```
>javac MyClass.java  
>java MyClass
```

The SWT libraries are available under the Eclipse plug-in directory. If you want to get the SWT libraries without downloading the whole Eclipse package, it is possible to download a single SWT library or the eclipse environment by clicking on the following link:

<http://www.eclipse.org/downloads/index.php>



Figure 5.1: Eclipse logo

Note that if it does not work with the preceding instructions, you can try the following actions (a bit more complex):

1. Put *swt.dll* in the same directory as the program.
2. Put *swt.dll* in the `JAVA_HOME\bin` directory.

```
>javac -classpath c:\swt\swt.jar MyClass.java
```

```
>java -classpath c:\swt\swt.jar;.
-Djava.library.path=c:\swt MyClass
```

The `java.library.path` is the required environment variable for the JNI. If you do not set this environment, your DLL class will not be accessible. In that case, the application cannot run properly and throws the run-time exception seen before in the chapter 3.

Chapter 6

SWT packages

SWT is composed of the following main packages. The definitions of the packages are taken from the Eclipse API documentation. You can find the API documentation on the Eclipse Web site:

<http://www.eclipse.org/documentation/html/plugins/org.eclipse.platform.doc.isv/doc/reference/api/>

org.eclipse.swt

Contains classes that define constants and exceptions used by the SWT classes. It consists of three classes: **SWT**, **SWTException** and, **SWTError**. The **SWT** will probably be your favorite class because it contains constants for SWT libraries such as keyboard, error, color, layout, text style, button,... and the style bit constants.

org.eclipse.swt.widgets

Contains most of the SWT widget components, including the support interface and classes. We will speak about them from the section 8.2.

org.eclipse.swt.events

Defines listeners and (typed) events, that SWT components use. This package contains three different groups of classes: the **Listener** interfaces, the **Adapter** and **Event** classes. We will speak about them in the section 8.1

org.eclipse.swt.dnd Contains classes for drag-and-drop support of the SWT widgets.

org.eclipse.swt.layout

Contains classes providing predefined positioning and sizing of the SWT widgets.

org.eclipse.swt.print

Contains classes providing print support for the SWT widgets.

org.eclipse.swt.graphics

Provides the classes concerning points, rectangles, regions, colors, cursors, fonts, graphics contexts (that is, GCs) where most of the primitive drawing operations are implemented, and images including both the code for displaying them and the public API for loading/saving them.

Chapter 7

Dialogs

What is and what does a dialog? A dialog is a kind of conversational window that appears when, e.g. the program needs to get informations from the user. It usually provides predefined solutions and the user only has to choose between them and confirm his choice. We will then have a look at the predefined existing Dialogs of SWT but first note that the dialog implementations are native. This means that the Dialogs – as the widgets – are platform components. They are derived from the Dialog abstract class. They are not Widgets but they can contain some of them.

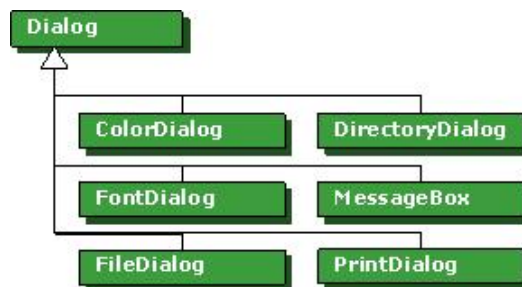


Figure 7.1: Dialog class hierarchy

There exist different types of dialogs: they can have specific properties (as shown here).

Each Dialog `open()` method returns a different type:

```
MessageBox messageBox = new MessageBox(shell, SWT.OK | SWT.CANCEL);  
if (messageBox.open() == SWT.OK)  
    System.out.println("Ok is pressed.");
```

Figure 7.2: Simple MessageBox code sample

- The *MessageBox* dialog returns an *int* from the `open()` method. Therefore, one must write different conditions to handle the return value for each Dialog. We will speak about them in the section 7.1.
- The *ColorDialog*, which shows a color selection palette, returns an RGB object from return method. We will speak about them in the section 7.2.
- The *DirectoryDialog*, which enables you to select a directory, returns a *String* from the `open()` method. The returning value is the selected directory. It is also possible to set additional filters to filter the directories. We will speak about them in the section 7.3.
- The *FontDialog*, which enables a user to select a font from all available fonts in the system, returns a *FontData* object from the `open()` method. We will speak about them in the section 7.4
- The *FileDialog* enables a user to select a file. Additionally, you can set an extension filter, path filter, and filename filters. This dialog has the following style bit constants: `SWT.OPEN` to show the Open button on the dialog and `SWT.SAVE` to show the Save button on the dialog.
- The *PrintDialog* enables a user to select a printer before printing. It returns a *PrinterData* object from the `open()` method.

Available button constants are listed below. A combination of different buttons can be made by using the `|` operator. The SWT framework builds the dialogs depending on their style bits. E.g. the button constants are: `SWT.ABORT`, `SWT.OK`, `SWT.CANCEL`, `SWT.RETRY`, `SWT.IGNORE`, `SWT.YES`, and `SWT.NO`.

The following table shows a list of available icons for the dialogs






SWT.ICON_ERROR	
SWT.ICON_INFORMATION	
SWT.ICON_QUESTION	
SWT.ICON_WARNING	
SWT.ICON_WORKING	

Figure 7.3: Most important style bit constants for the Dialogs for the icons

7.1 MessageBox

The *MessageBox* is really useful to inform or warn the user or to get a basic response from the user. It can display a message with an icon to the user and provide buttons to allow the user to respond to the system.

To add a *MessageBox* that asks the user something and gives him possible replies of "Yes", "No" or "Cancel", you can use the following code (the last part of this code will serve to treat the user's answer):

```
// Create a MessageBox with:
// - a Question icon
// - a Yes button
// - a No button
// - a Cancel button
MessageBox messageBox = new MessageBox(shell, SWT.ICON_QUESTION
    | SWT.YES | SWT.NO | SWT.CANCEL);
// Phrase in the MessageBox
messageBox.setMessage("Do you want something?");
// Title of the MessageBox
messageBox.setText("Question MessageBox");

// create an int to get the answer and treat it
int answer;

// open the MessageBox
answer = messageBox.open();

// treat the result which will happen
switch (answer) {
case SWT.YES:
    System.out.println("The user wants something");
    break;
case SWT.NO:
    System.out.println("The user does not want anything");
    break;
case SWT.CANCEL:
    System.out.println("The user cancelled");
}
```

Figure 7.4: MessageBox code sample



Figure 7.5: MessageBox resulting from the previous code

As said before, the `open()` method of the `MessageBox` class will return an *int* representing the response from the user.

The different icon styles were given in the figure 7.4 and the possible styles for the types of responses are: `SWT.OK`, `SWT.CANCEL`, `SWT.YES`, `SWT.NO`, `SWT.RETRY`, `SWT.ABORT`, `SWT.IGNORE`

7.2 ColorDialog

The *ColorDialog* allows users to select a color from a predefined set of available colors in a palette. To add a color dialog to your *Shell*, you can use the following code (the last part of this code will serve to treat the user's answer):

```
// Create a ColorDialog
ColorDialog colorDialog = new ColorDialog(shell);

// Title of the MessageBox
colorDialog.setText("My ColorDialog");

// to pre-select a color, e.g. the red
colorDialog.setRGB(new RGB(255, 0, 0));

// create an RGB object to get the answer and treat it
RGB answer = colorDialog.open();
System.out.println("Color selected: " + answer);
```

Figure 7.6: ColorDialog code sample



Figure 7.7: ColorDialog resulting from the previous code

As said before, the `open()` method of the *ColorDialog* class will return an object of type *RGB* representing the color the user selected. It will return null if the dialog was cancelled, if no color was selected or if an error occurred.

7.3 DirectoryDialog

The *DirectoryDialog* allows the user to navigate through the file system of his computer (or network) and select a directory. To add a *DirectoryDialog* to your *Shell*, you can use the following code (the last part of this code will serve to treat the user's answer):

```
// create a DirectoryDialog
DirectoryDialog directoryDialog = new DirectoryDialog(shell);

// set the title of the DirectoryDialog
directoryDialog.setText("My DirectoryDialog");

// to pre-select the directory where it will be opened at first.
directoryDialog.setFilterPath("F:/eclipse");

// create an String object to get the answer and treat it
String answer;
answer = directoryDialog.open();
System.out.println("Directory selected: " + answer);
```

Figure 7.8: DirectoryDialog code sample



Figure 7.9: DirectoryDialog resulting from the previous code

As said before, the `open()` method of the *DirectoryDialog* class will return a string describing the absolute path of the selected directory, or null if the dialog was cancelled or if an error occurred.

7.4 FontDialog

The *FontDialog* allows the user to select a font in all the available fonts in his system. To add a *FontDialog* to your *Shell*, you can use the following code (the last part of this code will serve to treat the user's answer):

```
// create a DirectoryDialog
DirectoryDialog directoryDialog = new DirectoryDialog(shell);

// set the title of the DirectoryDialog
directoryDialog.setText("My DirectoryDialog");

// to pre-select the directory where it will be opened at first.
directoryDialog.setFilterPath("F:/eclipse");

// create an String object to get the answer and treat it
String answer;
answer = directoryDialog.open();
System.out.println("Directory selected: " + answer);
```

Figure 7.10: FontDialog code sample



Figure 7.11: FontDialog resulting from the previous code

As said before, the `open()` method of the *FontDialog* class will return a *FontData* object describing the font that was selected, or null if the dialog was cancelled or an error occurred.

7.5 FileDialog

The *FileDialog* allows a user to navigate through the file system and select or enter a file name. To add a *FileDialog* to your shell you can use the following code (the last part of this code will serve to treat the user's answer):

```
// create a FileDialog
FileDialog fileDialog = new FileDialog(shell, SWT.OPEN);

// set the title of the FileBox
fileDialog.setText("My FileDialog to open");

// to pre-select the kind of file the dialog will show
String[] filterExtensions = {"*.txt", "*.doc", "*.*"};
fileDialog.setFilterExtensions(filterExtensions);

// create an String object to get the answer and treat it
String answer = fileDialog.open();
System.out.println("File selected: " + answer);
```

Figure 7.12: FileDialog code sample

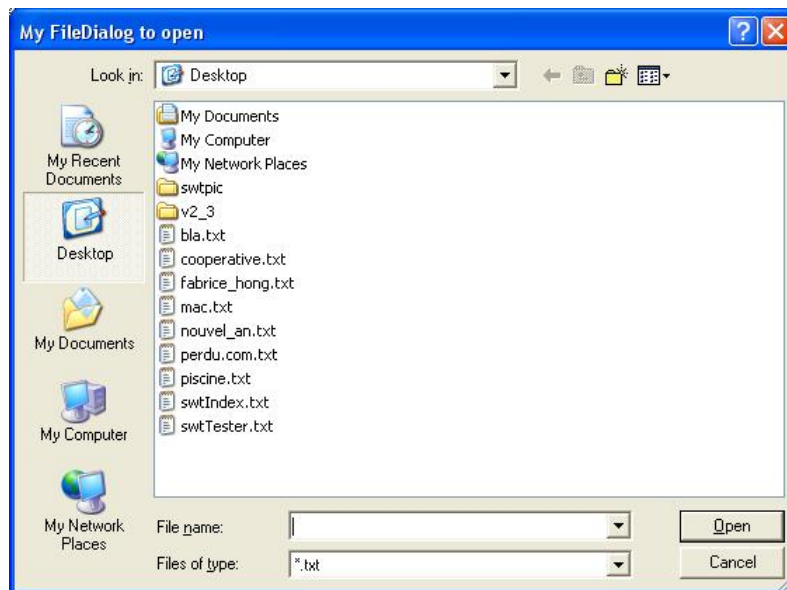


Figure 7.13: FileDialog resulting from the previous code

As said before, the `open()` method of the *FileDialog* class will return a *String* describing the absolute path of the first selected file, or null if the dialog was cancelled or an error occurred.

The possible styles of the FileDialog are `SWT.OPEN`, `SWT.SAVE` and `SWT.MULTI`

7.6 PrintDialog

The *PrintDialog* allows the user to select a printer and various print-related parameters prior to starting a print job. To add a *PrintDialog* to your *Shell* you can use the following code (the last part of this code will ser to treat the user's answer):

```
// create a PrintDialog
PrintDialog printDialog = new PrintDialog(shell, SWT.NONE);

// set the title of the PrintDialog
printDialog.setText("My PrintDialog");

// the scope of the print job will initially be set to print a range of
// pages, starting with second page and ending at the fifth page.
printDialog.setScope(PrinterData.PAGE_RANGE);
printDialog.setStartPage(2);
printDialog.setEndPage(5);

// the 'print to file' option will also be checked.
printDialog.setPrintToFile(true);

// create an PrinterData object to get the answer and treat it
PrinterData answer = printDialog.open();
if (answer != null) {
    switch (answer.scope) {
        case PrinterData.ALL_PAGES:
            System.out.println("Printing all pages.");
            break;
        case PrinterData.SELECTION:
            System.out.println("Printing selected page.");
            break;
        case PrinterData.PAGE_RANGE:
            System.out.print("Printing page range. ");
            System.out.print("From: " + answer.startPage);
            System.out.println(" To: " + answer.endPage);
            break;
    }
}
```

Figure 7.14: PrintDialog code sample (1/2)

```
    if (answer.printToFile) {  
        System.out.println("Printing to file.");  
    } else {  
        System.out.println("Not printing to file.");  
    }  
  
    if (answer.collate) {  
        System.out.println("Collating.");  
    } else {  
        System.out.println("Not collating");  
    }  
  
    System.out.println("Number of copies: " + answer.copyCount);  
    System.out.println("Printer name: " + answer.name);  
}
```

Figure 7.15: PrintDialog code sample (2/2)

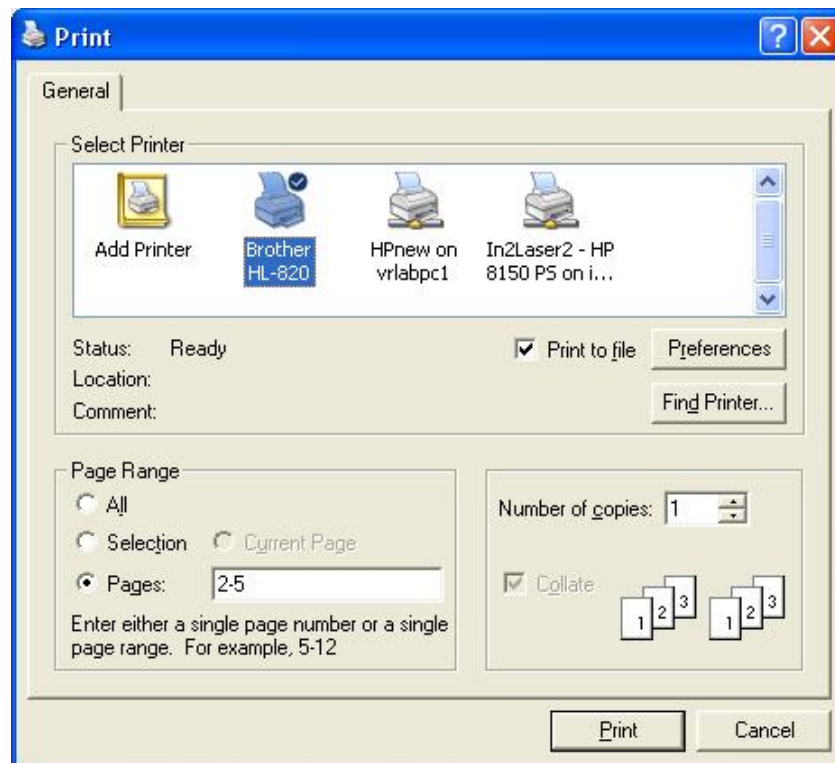


Figure 7.16: PrintDialog resulting from the preceding code

As said before, the `open()` method of the `PrintDialog` class will return a `PrinterData` object containing all the information the user selected from the dialog. If the dialog was cancelled or an error occurred, the `open()` method will return null.

Note that the `PrintDialog` class contains the following methods: `getEndPage()`, `getPrintToFile()`, `getScope()` and `getStartPage()` which return their respective fields the user selected before pressing OK or Cancel in the dialog. It is not recommended to use these methods to determine information for a print job as they return values even if the user cancelled the dialog.

Chapter 8

Widgets

What is and what does a widget? It is a combination of a graphic symbol and some program code to perform a specific function. E.g. a scroll-bar or button. Windowing systems usually provide widget libraries containing commonly used widgets drawn in a certain style and with consistent behaviour. We will see the most used of them but first, let us have a look at the hierarchy of the widgets to have a global idea of what we will explore: The *Menu* objects also require a window to be displayed, but this requirement is indirectly satisfied. A *Menu* object requires a *Control* object.

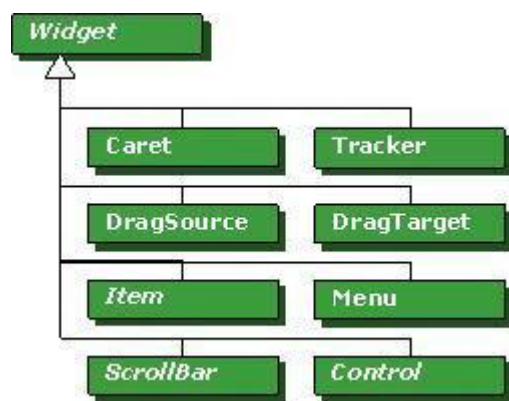


Figure 8.1: Widget class hierarchy

Note that the *Widget*, *Item*, *ScrollBar* and *Control* classes are abstract classes.

8.1 Widget Events

The Events and Listener are the widgets we use to manage with the user actions. They are not graphically visible, but without them we cannot verify the changes made by the user on the interface. You can find an *Event* widgets summary in the next table that, in order to simplify, only contains the event names. You should figure out the name of an event class by using this `<Event_Name>Event`. In the same idea, the name of the associated listener can be figured out by using `<Event_Name>Listener`. But pay attention, each event does not have a matching adapter class. This is why the events having adapters are marked in bold (the name of an adapter can be figured out by using `<EventName>Adapter`).

For example: the event name is *Control*, event class is *ControlEvent*, listener class is *ControlListener*, adaptor class is *ControlAdaptor*.

Event Name	Widgets	Generated When
Arm	MenuItem	a menu item is highlighted
Control	Control, TableColumn, Tracker	a control is resized or moved
Dispose	Widget	a widget is disposed
Focus	Control	a control gains or loses focus
Help	Control, Menu, MenuItem	the user requests help (e.g. by pressing the F1 key)
Key	Control	a key is pressed or released when the control has keyboard focus
Menu	Menu	a menu is hidden or shown
Modify	Combo, Text	a widget's text is modified
Mouse	Control	the mouse is pressed, released, or double-clicked over the control
MouseMove	Control	the mouse moves over the control
MouseTrack	Control	the mouse enters, leaves, or hovers over the control
Paint	Control	a control needs to be repainted
Selection	Button, Combo, CoolItem, List, MenuItem, Sash, Scale, ScrollBar, Slider, StyledText, TabFolder, Table, TableColumn, TableTree, Text, ToolItem, Tree	an item is selected in the control
Shell	Shell	the shell is minimized, maximized, activated, deactivated, or closed
Traverse	Control	the control is traversed (tabbed)
Tree	Tree, TableTree	a tree item is collapsed or expanded
Verify	Text, StyledText	a widget's text is about to be modified

Figure 8.2: Event widgets summary

8.1.1 How to use the listeners (SelectionListener)

When you want to capture the events in SWT, you need to add listeners to your controls. Then when the event corresponding to the listened widget occurs, the listener code will be executed.

To make this possible, you need to do two things:

1. First, you will have to create a listener which is specific to the event that you wish to capture. There is an interface for each listener that you can use: **SelectionListener** and a class that will provide you with the event informations: **SelectionEvent**. Within the listener that you created, you will have to implement the methods that are predefined in the interface (which appears automatically when you create a new object of this type if you use Eclipse).

```
SelectionListener listener = new SelectionListener() {  
    public void widgetSelected(SelectionEvent arg0) {  
        System.out.println("Button Selected");  
    }  
  
    public void widgetDefaultSelected(SelectionEvent arg0) {  
    }  
};
```

Figure 8.3: SelectionListener sample code

When it only requires to implement one method, you can create and use an adapter instead of a listener: **SelectionAdapter**. The adapter implements the interface with empty methods so that you do not have to write some code for methods that you do not need in your listener. Note that in our **SelectionListener**, we did not add any code for the **widgetDefaultSelected()** method. So we simply could have used an adapter, and only implement the **widgetSelected()** method.

```
SelectionAdapter adapter = new SelectionAdapter() {  
    public void widgetSelected(SelectionEvent arg0) {  
        System.out.println("Button Selected");  
    }  
};
```

Figure 8.4: SelectionAdapter sample code

2. The second thing is that you will have to add the listener to your widget. For this, you will see that each control has methods to add listeners: `addSelectionListener()`. Then you pass your listener as argument to this method, and you are ready to capture events.

```
Button button = new Button(shell, SWT.PUSH);  
button.addSelectionListener(listener);
```

Figure 8.5: ButtonListener sample code

Or – in our case – we could have also used:

```
button.addSelectionListener(adapter);
```

Figure 8.6: ButtonAdapter sample code

Now that we have seen how to create, add and use a listener to a widget, let us have a look at the most often used types of listeners.

8.1.2 KeyListener

The *KeyListener* object has two methods: `keyPressed()` and `keyReleased()`, whose functions are obvious. The event object is of *KeyEvent* type and because there are only two methods, a *KeyAdapter* can also be used to implement only one of these methods.

The *KeyEvent* is the most interesting feature to capture keyboard events. It has a property called `character` that stores which character was just pressed. You can access this character value using `e.character` (where `e` is a *KeyEvent* instance). You can also access the key code and state mask of the pressed key. The key codes are for special keys and are constants in the SWT library, like `SWT.CTRL` for the control key and `SWT.CR` for the carriage return key and the state mask indicates the status of the keyboard modifier keys.

The following example shows how to use the `keyPressed()` and `keyReleased()` methods. At first, we attach a *KeyListener* to a basic widget (e.g. a *Text*) and a *KeyEvent* is generated every time you type in the text. With the `keyPressed()` method, we check if the pressed key was a special key (such as tab, carriage return, etc.), and print out the readable character for that key. Otherwise, we print out the character as is. In the `keyReleased()`

method, we check if the control key was pressed before another character was pressed, such as *Control-C*. This is a method that could be used to capture *ctrl-key* sequences in a widget such as a text:

```
Text text = new Text(shell, SWT.MULTI | SWT.BORDER);
text.setBounds(10, 10, 100, 100);
text.addKeyListener(new KeyListener() {
    public void keyPressed(KeyEvent e) {
        String string = "";
        switch (e.character) {
            case 0:
                string += " '\\0'";
                break;
            case SWT.BS:
                string += " '\\b'";
                break;
            case SWT.CR:
                string += " '\\r'";
                break;
            case SWT.DEL:
                string += " DEL";
                break;
            case SWT.ESC:
                string += " ESC";
                break;
            case SWT.LF:
                string += " '\\n'";
                break;
            default:
                string += " '" + e.character + "'";
                break;
        }
        System.out.println(string);
    }

    public void keyReleased(KeyEvent e) {
        if (e.stateMask == SWT.CTRL && e.keyCode != SWT.CTRL)
            System.out.println("Command can execute here");
    }
});
```

Figure 8.7: KeyListener code sample

Note that if we used a KeyAdapter instead of a KeyListener, we would have only had to implement one of `keyPressed()` and `keyReleased()` method.

8.1.3 **MouseListener**

There are three kinds of *MouseListener*:

1. *MouseListener*, that tracks the mouse pression
2. *MouseMoveListener*, that tracks when the mouse is moved
3. *MouseTrackListener*, that tracks where the mouse is in relation with the *Control*

We will go over all these three listeners. But since we are dealing with events, it is not possible to show the results.

MouseListener

The *MouseListener* has three methods, `mouseDown()`, `mouseUp()` and the `mouseDoubleClick()` (obvious functions). Once again a *MouseAdapter* is available and *MouseEvent* is the class that contains information on the event. The *MouseEvent* fields are `x`, `y`, `stateMask` and `button`. This last property returns the number of the button that was pressed or released: a value of 1 for the first button on the shell, 2 for the the second,... The `x` and `y` properties return the mouse coordinates at the moment of the event and the `stateMask` property returns the state of the keyboard modifier keys at the event time.

In the following example there are two *MouseAdapters*:

- one that implements the `mouseUp()` and `mouseDown()` methods (this adapter is attached to two different buttons. We can do this by first creating the adapter, then adding it to every widgets that needs to use it. We find the number of the button that has been pressed, and also the coordinates of the mouse when it is both pressed and released)
- another one that implements the `mouseDoubleClick()` method (it opens a new window when the double click event is detected on the *Label*)

```
Button button = new Button(shell, SWT.PUSH);
button.setText("Push Me");
button.setBounds(10, 10, 60, 20);
Button button2 = new Button(shell, SWT.PUSH);
button.setText("Push Me Too");
button.setBounds(100, 10, 60, 20);
MouseAdapter mouseAdapter = new MouseAdapter() {
    public void mouseDown(MouseEvent e) {
        System.out.println("Button " + e.button + " pressed");
        System.out
            .println("Mouse pressed at (" + e.x + "," + e.y + ")");
    }

    public void mouseUp(MouseEvent e) {
        System.out.println("Mouse released at (" + e.x + "," + e.y
            + ")");
    }
};
button.addMouseListener(mouseAdapter);
button2.addMouseListener(mouseAdapter);
Label label = new Label(shell, SWT.NONE);
label.setText("Double Click Me");
label.setBounds(10, 40, 100, 20);
label.addMouseListener(new MouseAdapter() {
    public void mouseDoubleClick(MouseEvent e) {
        Shell shell2 = new Shell(display);
        shell2.setSize(100, 100);
        Label label2 = new Label(shell2, SWT.NONE);
        label2.setText("Hello New Window!");
        label2.setBounds(0, 50, 100, 20);
        shell2.open();
    }
});
```

Figure 8.8: MouseListener code sample

Note that if you press on the first button, it tells you that Button 1 has been pressed, and pressing on the second button tells you that Button 2 has been pressed. If you press your mouse down, move it around, and then release it, anywhere else you will see the different coordinates where your mouse was pressed and released at.

MouseMoveListener

There is only one method for the *MouseMoveListener*: `mouseMove()`. This method is invoked when the mouse is moved over the control onto which the listener is attached. As usual, *MouseEvent* is the type of the generated

event.

MouseListener

The *MouseListener* has three methods: `mouseEnter()` that is called when the mouse enters into the area covered by the Control to which it is attached, `mouseExit()` that is called when the mouse exits the area covered by the Control to which it is attached and `mouseHover` that is called when the mouse hovers over the area covered by the Control to which it is attached.

8.1.4 TextListener

There are a couple of listeners that are very useful when you use the *Text* widgets: *ModifyListener* which is called when text is modified and *VerifyListener* which is called before text is modified. The last one can be used to verify that the new entered values are valid. We will briefly go over both of these listeners.

ModifyListener

The *ModifyListener* has one method, `modifyText()` whose produced event is a *ModifyEvent* (similar to *TypedEvent*, and has a `time` and a `widget` fields. It can be used when we need to be notified if the text has been changed. The `modifyText()` method is called every time a character is modified (add or erased). You can see an example of its used in the end of the *VerifyListener* subsection.

VerifyListener

The only method of the *VerifyListener* is `verifyText()` and its event is of type *VerifyEvent*. This event has all the properties of a *TypedEvent* and a *KeyEvent*, as well as the fields: `start` that indicate the range of text which is about to be modified, `end` that indicate the range of text which is about to be modified, `doit` that indicates whether or not we should complete the action that triggered the event and `text` that indicates the new text to which your text is being changed.

We can use this listener to be sure that a field can be filled by some specific characters. In the following example, we check if the user is entering an asterisk (*) in the *Text* widget and if it is the case the action is cancelled. (Notice that it does not matter how many times the "*" key is pressed... no asterisk will never appear in the widget.

```
Text text = new Text(shell, SWT.MULTI | SWT.WRAP);
text.setBounds(10, 10, 200, 100);
text.setText("Here is some sample text");
text.addModifyListener(new ModifyListener() {
    public void modifyText(ModifyEvent e) {
        System.out.println("Modified at " + e.time);
    }
});
text.addVerifyListener(new VerifyListener() {
    public void verifyText(VerifyEvent e) {
        if (e.text.equals("*")) {
            System.out.println("Cannot type *");
            e.doit = false;
        }
    }
});
```

Figure 8.9: VerifyListener code sample

8.1.5 FocusListeners and TraverseListener

As usual, *FocusListeners* are in relation with the *FocusEvent* that are generated when a widget receives the focus, by any means. And by the same way, *TraverseListeners* are in relation with *TraverseEvent* which are generated when a widget gains the focus by means of traversal. Let us see both of these event handlers.

FocusListener

The *FocusListener* has two methods: `focusGained()` and `focusLost()` whose functions seem to be obvious. The generated event is of type *FocusEvent* and has all the properties of a *TypedEvent*. The example in the *TraverseListener* paragraph will give you more details.

TraverseListener

The only method of the *TraverseListener* is `keyTraversed()`. It is called when the widget to which the handler is attached is traversed by a traversal method (e.g. "tabs", "up arrow", "down arrows",...).

The generated event is the *TraverseEvent* whose the most useful properties are: `detail` that indicates what kind of traversal generated the event and `doit` that is the same as for the *VerifyListener* (it sets a flag indicating whether the event should continue).

All these traversals can be referred to by using the `SWT.TRAVERSE.XXX` style

bit constants (e.g. `SWT.TRAVERSE_TAB_PREVIOUS`).

```
Button b1 = new Button(shell, SWT.PUSH);
Button b2 = new Button(shell, SWT.PUSH);
Button b3 = new Button(shell, SWT.PUSH);
Button b4 = new Button(shell, SWT.PUSH);
Button b5 = new Button(shell, SWT.PUSH);
Button b6 = new Button(shell, SWT.PUSH);
b1.setBounds(10, 10, 50, 50);
b2.setBounds(100, 10, 50, 50);
b3.setBounds(200, 10, 50, 50);
b4.setBounds(10, 100, 50, 50);
b5.setBounds(100, 100, 50, 50);
b6.setBounds(200, 100, 50, 50);
b1.setText("1");
b2.setText("2");
b3.setText("3");
b4.setText("4");
b5.setText("5");
b6.setText("6");
FocusListener focusListener = new FocusListener() {
    public void focusGained(FocusEvent e) {
        System.out.println(e.widget + " has focus");
    }

    public void focusLost(FocusEvent e) {
        System.out.println("And now has lost it.");
    }
};
```

Figure 8.10: TraverseListener code sample

8.2 Useful widgets

Here is a hierarchy of the *Control* widgets you can use:

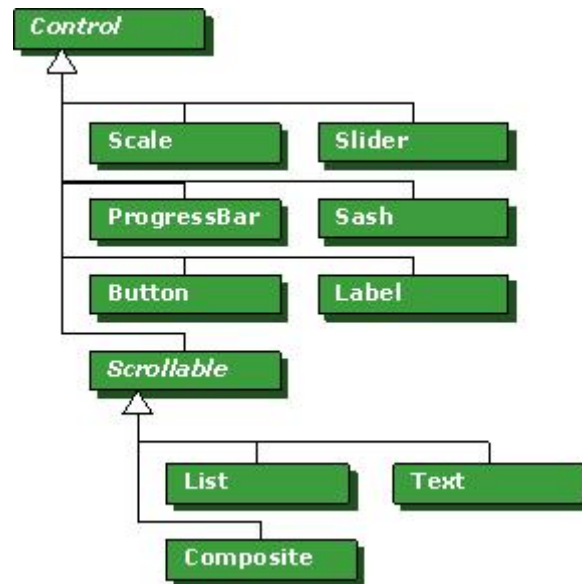


Figure 8.11: Control class hierarchy

*You always need to specify at least a style bit constant for every one of them (even **SWT.NONE** if you do not know which constant to use or if you do not want to specify anyone).*

8.2.1 Buttons

A *Button* is a widget the user can click onto in order to start an action. To place a *Button* on the *Shell*, you can use the following code:

```
// create a Button of PUSH-style
Button button = new Button(shell, SWT.PUSH);

// set the text of the button
button.setText("OK");

// set the location of the button
button.setLocation(10, 10);

// set the size of the button
button.setSize(50, 20);
```

Figure 8.12: Button code sample

And here is the result of the previous code:



Figure 8.13: Button resulting from the previous code

A *Button* can have different styles that depend on its defined style bit constant(s). A list of Button objects and their style constants is shown in table on the following page.


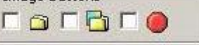



Constants	Example	Description
SWT.ARROW		A button to show popup menus etc. Direction of the arrow is determined by the alignment constants.
SWT.CHECK	<div>Text Buttons</div> <input type="checkbox"/> One <input type="checkbox"/> Two <input type="checkbox"/> Three <div>Image Buttons</div> 	Check boxes can have images as well.
SWT.PUSH	<div>Text Buttons</div> <input type="button" value="One"/> <input type="button" value="Two"/> <input type="button" value="Three"/> <div>Image Buttons</div> 	A push button.
SWT.RADIO	<div>Text Buttons</div> <input type="radio"/> One <input type="radio"/> Two <input type="radio"/> Three <div>Image Buttons</div> 	Radio buttons can be used in a group.
SWT.TOGGLE	<div>Text Buttons</div> <input type="button" value="One"/> <input type="button" value="Two"/> <input type="button" value="Three"/> <div>Image Buttons</div> 	Like SWT.PUSH, but it remains pressed until a second click.

Figure 8.14: Useful button style bit constants

A basic event handler for the button is a selection event handler which is called when the button is selected by clicking on it. To create the event handler, we add a listener to the button using the preceding code as seen in the section 8.1.1

8.2.2 Slider, Scale and ProgressBar widgets

The *Slider*, *Scale* and *ProgressBar* are all very similar *Control* widgets, but they do not really use the same methods in order to function properly.

Slider and Scale

The *Scale* widgets represent a range of selectable continue values. This range can be defined by the `setMinimum()` and `setMaximum()` methods. You can get the selection value by using the `getSelection()` method (and by the same way, set it with the `setSelection()` method). A *Scale* widget can

only have one selected value at each time.

If you use a *Slider*, you can also set the size of the thumb. This size is relative to the minimum and maximum values that you have set. For example, if you set the minimum to 0, the maximum to 100, and the thumb size to 30 – with the `setThumb()` method –, you will make the thumb of about one third (100 divided 30 equals almost 1/3) of the length of the slider.

If you use a *Scale*, you can set the page increment that determines how often measuring bars are placed on the scale. For example, if you set the minimum to 0, the maximum to 500 and the page increment to 50 – with the `setPageIncrement()` method –, there will be 10 (500 divided by 50 equals 10) measuring bars on the scale. It will place a bar every 50 places, relative to the minimum and the maximum. It is so impossible to have several selections. Here is an example of what you get with the following code:

```
Scale scale1 = new Scale(shell, SWT.HORIZONTAL | SWT.BORDER);
scale1.setBounds(10, 10, 200, 40);
scale1.setMinimum(0);
scale1.setMaximum(500);
scale1.setSelection(100);
scale1.setPageIncrement(50);

Slider slider1 = new Slider(shell, SWT.VERTICAL);
slider1.setBounds(240, 10, 20, 150);
slider1.setSelection(50);
slider1.setMaximum(100);
slider1.setMinimum(0);
slider1.setThumb(30);
```

Figure 8.15: Scale and Slider code sample

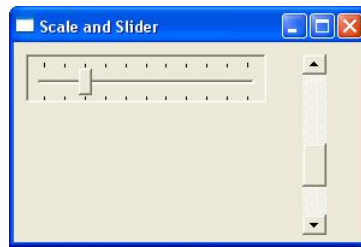


Figure 8.16: Scale and Slider resulting from the previous code

We can change the orientation of the *Scale* and *Slider* widgets depending on the style bit constants we use in the constructor (`SWT.VERTICAL` or `SWT.HORIZONTAL`).

Optionally, you can use the `SWT.BORDER` constant to create a border around the Scale widget, but this will not have any effect on the Slider widget.

ProgressBar

The main difference between the *ProgressBar* widgets and the *Slider* and *Scale* widgets is that you cannot select it. It usually shows the progress of a task and its principle style bit constants are: `SWT.HORIZONTAL` and `SWT.VERTICAL` for the orientation, `SWT.SMOOTH` that makes a continue progress line and `SWT.INDETERMINATE` that makes the progress line always get bigger – as for some downloads under the Windows OS.

Here is a simple code example and its result:

```
ProgressBar progressBar1 = new ProgressBar(shell, SWT.HORIZONTAL);
progressBar1.setMinimum(0);
progressBar1.setMaximum(100);
progressBar1.setSelection(30);
progressBar1.setBounds(10, 40, 200, 20);

ProgressBar progressBar2 = new ProgressBar(shell, SWT.HORIZONTAL
    | SWT.SMOOTH);
progressBar2.setMinimum(0);
progressBar2.setMaximum(100);
progressBar2.setSelection(30);
progressBar2.setBounds(10, 100, 200, 20);
```

Figure 8.17: ProgressBar code sample

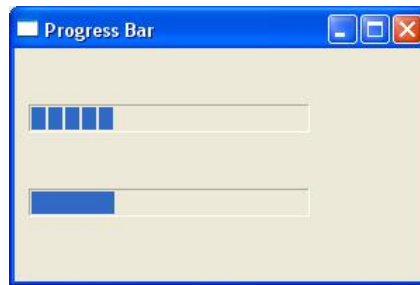


Figure 8.18: ProgressBar resulting from the previous code

8.2.3 Text widget

A *Text* widget can be used to show or to edit text. You can also use a *Styled-Text* widget if you want to display a coloured (characters or background) text with different fonts and sizes.

To create your *Text* widget, you can use the following style bit constants: `SWT.MULTI` or `SWT.SINGLE` to have a multi or single line widget, `SWT.READ_ONLY`, `SWT.WRAP` to break the lines at the end of the widget and, as the *Text* widget is a scrollable widget, `SWT.H_SCROLL` and `SWT.V_SCROLL` to add horizontal or vertical scrollbars to your *Text* widget.

Furthermore the *Text* widget has some interesting properties that can be set. For example, if your text is editable, you can set the maximum number of characters that can be put in by using the `setTextLimit()` method. You can also create some kind of password text boxes by using the `setEchoChar()` method with an asterisk character as argument.

The commented code on the following page shows how to create the three specific kind of Text widgets appearing in the next window:

```
// wrapped, bordered and read-only text with multiples lines and a
// vertical scrollbar
Text wrappedReadOnlyMultiVScrollBorderText = new Text(shell, SWT.WRAP
    | SWT.READ_ONLY | SWT.MULTI | SWT.V_SCROLL | SWT.BORDER);
wrappedReadOnlyMultiVScrollBorderText
    .setText("Hello, this is a wrapped and bordered text field" +
        "with multiple lines and a vertical scrollbar");
wrappedReadOnlyMultiVScrollBorderText.setLocation(10, 10);
wrappedReadOnlyMultiVScrollBorderText.setSize(200, 50);

// text with a single line and an horizontal scrollbar
Text singleHScrollText = new Text(shell, SWT.H_SCROLL);
singleHScrollText
    .setText("Hello, this is a text field on a simple line " +
        "with an horizontal scrollbar");
singleHScrollText.setLocation(10, 70);
singleHScrollText.setSize(200, 30);

// bordered text for user to enter a password with a max of 10
// characters
Text borderedText = new Text(shell, SWT.BORDER);
borderedText.setText("password");
borderedText.setTextLimit(10);
borderedText.setEchoChar('*');
borderedText.setLocation(10, 110);
borderedText.setSize(200, 20);
```

Figure 8.19: Text sample code



Figure 8.20: Text resulting from the previous code

8.2.4 List widget

The *List* widget is usually used to display a list of string values that sends a notification event to its listener when the user selects one of them. This selection can be single or multiple, depending on the style bit constant used: `SWT.SINGLE` or `SWT.MULTI`. And as the *List* is a *Scrollable* widget, you can add scrollbars by using the `SWT.H_SCROLL` or `SWT.V_SCROLL` style bit constant. And you can obviously add items to a list, by using the `setItems()` or `add()` method.

In the following example, two lists are created. The first list uses the `SWT.H_SCROLL` style to create an horizontal scrollbar within the list. The second one uses a *MouseListener* to respond to the `MouseDown()` and the `MouseUp()` methods, as seen in the subsection 8.1.3.

This is the code that is necessary to get the result on the following page:

```
List list1 = new List(shell, SWT.MULTI | SWT.H_SCROLL);
list1.setItems(new String[] { "Strawberry", "Banana", "Apple" });
list1.add("Pickle");
list1.setBounds(0, 0, 60, 100);

final List list2 = new List(shell, SWT.SINGLE | SWT.BORDER);
list2.setItems(new String[] { "Rock", "Paper", "Scissors" });
list2.setBounds(110, 0, 50, 50);
list2.addMouseListener(new MouseAdapter() {
    public void mouseDown(MouseEvent e) {
        System.out.println(list2.getSelection()[0] + " wins");
    }

    public void mouseUp(MouseEvent e) {
        System.out.println("Try again!");
    }
});
```

Figure 8.21: List sample code

Note that list2 must be declared as final in order to use it within the adapter.

And here is the result of the previous code:



Figure 8.22: List resulting from the previous code

8.3 Composite widgets

As said in the previous chapters, a *Composite* widgets can contain other *Composite* widgets which would be placed inside of the *Composite* by the same way as the widgets are placed on a *Shell*. Then every widget position inside the *Composite* is relative to this last one, so if this *Composite* is moved on the *Shell*, the widgets inside of it keep their relative positions.

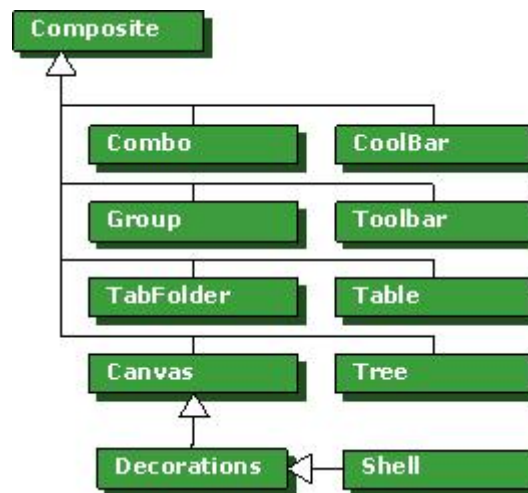


Figure 8.23: Composite can contain other composite classes

The *Composite* classes can usually contain other composite classes. This containment is built up by using the constructor of a *Composite* widget class. But in contrary to the Swing GUI, there is not an `add()` method. So you must use their constructors to build up a containment structure.

As you can see in the previous figure, the *Shell* class is also a *Composite* class. This means that the *Shell* object can contain other *Composite* classes. The *Composite* widgets are *Scrollable*, it is so possible to add scrollbars to the *Composite* widgets by using the `SWT.H_SCROLL` or `SWT.V_SCROLL` style bit constants. Furthermore, composite properties as the background colour can be set except the text value.

The following code is an example that shows how to place a composite within a *Composite*. The first *Composite* has a border and a coloured background, so it is easily spotted on the screen. In order to use the `setBackground()` method for any control, you need to import the Graphics library and create a `Color`:

```
Composite composite1 = new Composite(shell, SWT.BORDER);
composite1.setBounds(10, 10, 270, 250);
composite1.setBackground(new Color(display, 31, 133, 31));
Label label = new Label(composite1, SWT.NONE);
label.setText("Here is a green composite");
label.setBounds(10, 10, 200, 20);
Composite composite2 = new Composite(composite1, SWT.H_SCROLL
    | SWT.V_SCROLL);
composite2.setBounds(10, 40, 200, 200);
List list = new List(composite2, SWT.MULTI);
for (int i = 0; i < 50; i++) {
    list.add("Item " + i);
}
list.setSize(300, 300);
```

Figure 8.24: Composite code sample

And this will produce the following result on the next page:

But first, note that if you do not give a border or a background to the composite, you will not be able to distinguish it from the shell. The second composite exists within the first. It has horizontal and vertical scrollbars, and a list within the composite

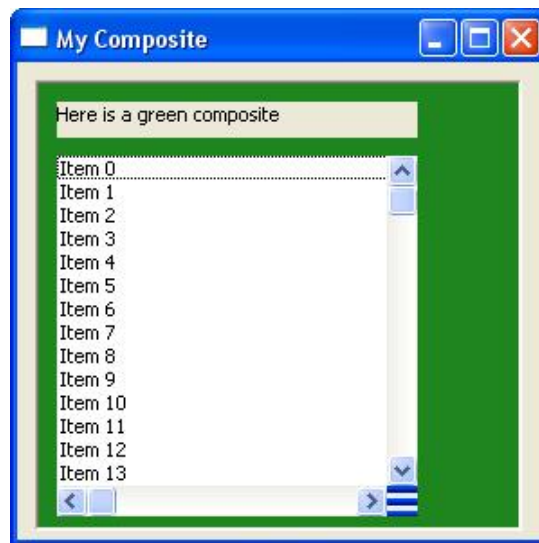


Figure 8.25: Composite resulting from the previous code

8.3.1 Table widget

The *Table* widget usually displays a set of *String* items or *Images*. With a *Table*, two options can be set: the lines and the column headers visibility. Both of these properties are set as default to false. You will see how to set these properties with the following source code. Note that in contrast to other *Composite* widgets, adding *Composite* controls to the *Table* widget is not possible. The *Table* sub-component must be of type *TableItem*.

The following style bit constants can be used: `SWT.SINGLE` or `SWT.MULTI` to enable single or multiple selection, `SWT.FULL_SELECTION` to enable the full row selection, `SWT.CHECK` to display a checkbox at the beginning of each row and obviously `SWT.V_SCROLL` and `SWT.H_SCROLL`.

Let's see now how to construct a *Table*:

```
// create the Table
Table table1 = new Table(shell, SWT.BORDER);
// set the location
table1.setBounds(10, 10, 270, 60);
// set the lines of the table visible
table1.setLinesVisible(true);
// set the headers of the table visible
table1.setHeaderVisible(true);
```

Figure 8.26: Table code sample

Once the *Table* is created, it only has one column. And if you try to add any items to the table, only the first field of the item will be displayed because it is the wrong way to do it. In order to display all the fields you want, you need to define *TableColumns* objects.

TableColumn

When you add *TableColumns* to your *Table*, you can use the following style bit constants: `SWT.LEFT`, `SWT.RIGHT` and `SWT.CENTER`.

```
// set the TableColumns
// create the TableColumns
TableColumn name = new TableColumn(table1, SWT.LEFT);
// set the name of the column
name.setText("Name");
// set the width of the column
name.setWidth(50);
```

Figure 8.27: TableColumn code sample

Do not forget to set the column width, and its name if you want to display the headers with the `setWidth()` and `setText()` methods. After you have created and added the columns to your table, you are now ready to add the *TableItems* objects.

TableItem

When you add a *TableItem* to your *Table*, you can not use any style bit constant (except `SWT.NONE` if you want):

```
// set the TableItems
// create the TableItem
TableItem item1 = new TableItem(table1, SWT.NONE);
// set the fields of the Item
item1.setText(new String[] { "Sarah", "15", "390 Sussex Ave" });
```

Figure 8.28: TableItem code sample

To set the text of an item, you can use the `setText()` method that will create a *TableItem* with fields in the order that you defined them. If you want to define one field at a time in a special order, you can use the `setText(int, string)` method which places the text into a specific field in the record.

And now that you have created *TableColumns* and *TableItems*, your table is full.

Let us have a look at three simple examples of tables and see the different styles that can be used.

- The first table – just below – has a border, fixed size columns, with different horizontal alignments.

```
// First table
// create the Table
Table table1 = new Table(shell, SWT.BORDER);
table1.setBounds(10, 10, 270, 60);
table1.setLinesVisible(true);

// set the TableColumns
TableColumn name = new TableColumn(table1, SWT.LEFT);
name.setText("Name");
name.setWidth(50);

TableColumn age = new TableColumn(table1, SWT.RIGHT);
age.setText("Age");
age.setWidth(30);

TableColumn address = new TableColumn(table1, SWT.LEFT);
address.setText("Address");
address.setWidth(200);

// set the TableItems
TableItem item1 = new TableItem(table1, SWT.NONE);
item1.setText(new String[] { "Sarah", "15", "390 Sussex Ave" });

TableItem item2 = new TableItem(table1, SWT.NONE);
item2.setText(new String[] { "Joseph", "56", "7 Yourstreet St" });
```

Figure 8.29: Table source code example 1

- The second table – just below – has a `CheckBox` column. We can set the checked status of the item through the code. There are no grid lines, but the header is visible. This example also sets the background color of one of the table items.

```
// Second table
// create the table
Table table2 = new Table(shell, SWT.CHECK | SWT.HIDE_SELECTION);
table2.setBounds(10, 80, 270, 80);
table2.setHeaderVisible(true);

// set the TableColumns
TableColumn fruit = new TableColumn(table2, SWT.LEFT);
fruit.setText("Fruit");
fruit.setWidth(100);

TableColumn colour = new TableColumn(table2, SWT.LEFT);
colour.setText("Colour");
colour.setWidth(170);

//set the TableItems
TableItem fruit1 = new TableItem(table2, SWT.NONE);
fruit1.setText(0, "Apple");
fruit1.setText(1, "Red");
fruit1.setChecked(true);

TableItem fruit2 = new TableItem(table2, SWT.NONE);
fruit2.setText(new String[] { "Kiwi", "Green" });
fruit2.setBackground(new Color(display, 255, 0, 0));

TableItem fruit3 = new TableItem(table2, SWT.NONE);
fruit3.setText(new String[] { "Banana", "Yellow" });
```

Figure 8.30: Table source code example 2

- The last example – just below – uses the `FULL_SELECTION` style, which means that when you select a *TableItem*, the whole item is highlighted instead of just the first field. The first column is resizable (this means you can make it larger or smaller). Note that each of the three columns has a different horizontal alignment. This example also shows how you can pre-select an item.

```
// Third table
// create the Table
Table table3 = new Table(shell, SWT.FULL_SELECTION);
table3.setLinesVisible(true);
table3.setBounds(10, 180, 270, 80);

// set the TableColumns
TableColumn first = new TableColumn(table3, SWT.LEFT);
first.setResizable(true);
first.setText("First");
first.setWidth(80);

TableColumn second = new TableColumn(table3, SWT.CENTER);
second.setText("Second");
second.setWidth(80);

TableColumn third = new TableColumn(table3, SWT.RIGHT);
third.setText("Third");
third.setWidth(80);

// set the TableItems
String[] numbers = new String[] { "One", "Two", "Three" };

TableItem firstItem = new TableItem(table3, SWT.NONE);
firstItem.setText(numbers);

TableItem secondItem = new TableItem(table3, SWT.NONE);
secondItem.setText(numbers);

TableItem thirdItem = new TableItem(table3, SWT.NONE);
thirdItem.setText(numbers);

TableItem fourthItem = new TableItem(table3, SWT.NONE);
fourthItem.setText(numbers);

table3.select(1);
```

Figure 8.31: Table source code example 3

And the following window is the result of these three examples:



Figure 8.32: Table resulting from the previous codes

8.3.2 Combo widget

The *Combo* widget allows users to select or add a new value to a list of values. It is similar to the *List*, but uses a limited space.

If you want to place a Combo box on the *Shell* and add items to it, you can use the following code:

```
Combo combo1 = new Combo(shell, SWT.DROP_DOWN | SWT.READ_ONLY);
combo1.setItems(new String[] { "One", "Two", "Three" });
```

Figure 8.33: Simplest Combo code sample

But although the *Combo* widget is a *Composite*, it does not make any sense to add child elements to it if they are not of the type *String*. So, an element to a *Combo* widget can be added by using the `add(String element)` method. The following SWT style bit constants can be used: `SWT.READ_ONLY`, `SWT.SIMPLE` or `SWT.DROP_DOWN`.

Furthermore, if you want to add *Items* to a *Combo*, you can use the `setItems()` method which takes an Array of Strings as argument. And then to select one of the list items, you must use the `select()` method which takes an integer as argument – the index of the item you want to select. In the following example three Combos will be created. The first is a drop-down list whose location and size are set, the second a simple Combo whose location and size are set at one time using the `setBounds()` method and the third a Combo with no Items.

```
// create the first Combo
Combo combo1 = new Combo(shell, SWT.DROP_DOWN|SWT.READ_ONLY);
combo1.setItems(new String[] { "One", "Two", "Three" });
combo1.select(0);
combo1.setLocation(0,0);
combo1.setSize(100,20);

// create the second Combo
Combo combo2 = new Combo(shell, SWT.SIMPLE);
combo2.setItems(new String[] { "Red", "Green", "Blue", "Yellow" });
combo2.setBounds(50,50,200,150);
combo2.select(1);

// create the third Combo
Combo combo3 = new Combo(shell, SWT.DROP_DOWN);
combo3.setLocation(200,0);
combo3.setSize(50,50);
```

Figure 8.34: Combo code sample

And on the next page, you will see the result...



Figure 8.35: Combo resulting from the previous code

8.3.3 Tree

The *Tree* widgets are usually used to display informations (e.g. folders and sub-folders for a file manager or classes and sub-classes for a class browser) in a hierarchical format. And even if the *Tree* class is a *Composite*, you may not add *Composite* classes to it except if it is of type *TreeItem*.

For a tree, the standard style bit constants are: `SWT.MULTI` to allow multiple selections or `SWT.SINGLE` to allow the selection of only one item from the *Tree* at a time and `SWT.CHECK` to add check boxes beside each item in the tree. (As usual, you cannot specify `SWT.SINGLE` and `SWT.MULTI` at the same time.)

The Tree we will create here with the following pieces of code allows multiple selections, and has a border. As usual, the first parameter of the *Tree* constructor is the *Composite* on which we want to place the tree, and the second is the SWT style bit constants.

```
final Tree tree = new Tree(shell, SWT.MULTI | SWT.BORDER);
tree.setSize(400, 300);
tree.setLocation(5, 5);
```

Figure 8.36: Tree code sample

Once the *Tree* is created, we can now attach *TreeItems* to it. We will here create the top-level *TreeItem* (i.e. the root). For this, the first parameter to the constructor is the *Tree* that we want to attach the items to, the second is the SWT style bit constant, and the (optional) last parameter is the index at which to place the item in the tree. In the following code sample, you will see how the first argument specifies that "My Documents" will appear as the middle element in the top-level of the tree, even it was added after the other two items.

```
TreeItem myComp = new TreeItem(tree, SWT.NONE);
myComp.setText("My Computer");
TreeItem netPlaces = new TreeItem(tree, SWT.NONE);
netPlaces.setText("My Network Places");
TreeItem myDocs = new TreeItem(tree, SWT.NONE, 1);
myDocs.setText("My Documents");
```

Figure 8.37: *TreeItem* source code example 1

Then if you want to create children *TreeItems* out of these root-level nodes, you can use a similar constructor, except that this time, instead of specifying the *Tree* to attach them to, you will specify the *TreeItem* so that they will appear under. In our code sample, both of these nodes will appear under the "My Computer" *TreeItem*. As in the above case, we can use the optional last parameter to specify the index at which the new element should be placed in the sub-tree.

```
TreeItem hardDisk = new TreeItem(myComp, SWT.NONE);
hardDisk.setText("Local Disk (C:)");
TreeItem floppy = new TreeItem(myComp, SWT.NONE, 0);
floppy.setText("3.5 Floppy (A:)");
```

Figure 8.38: *TreeItem* source code example 2

You can then continue the creating sub-trees process as many times as you want only by creating more *TreeItems* out of other *TreeItems*. Here we extend the *Tree* to a third level by creating a *TreeItem* out of the "Local Disk" *TreeItem*.

```
TreeItem progFiles = new TreeItem(hardDisk, SWT.NONE);
progFiles.setText("Program Files");
```

Note that we could also add a different image to each *TreeItem* by loading the image and then using *TreeItems* `setImage()` method, as follows where the image will appear to the left of the *Item* text.

```
Image icon = new Image(display, "test.bmp");
progFiles.setImage(icon);
```

Figure 8.39: TreeImage code sample

The next step is to add a *Listener* to the tree. We will associate our *SelectionListener* with the *Tree* as a whole, and so we will be allowed to use the `getSelection()` method to return a list of the selected *TreeItems*. In the example below, we are listening for changes in the selection(s), to get an array of the *TreeItems* that are selected and print out the text from each item. (For more informations on the *SelectionListener*, have a look to the section 8.1.1)

```
tree.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent e) {
        TreeItem[] t = tree.getSelection();
        System.out.print("Selection: ");
        for (int i = 0; i < t.length; i++) {
            System.out.print(t[i].getText() + ", ");
        }
        System.out.println();
    }
});
```

Figure 8.40: TreeSelectionListener code sample

And we can also listen for collapsing and expanding of the tree using a *TreeListener*. We will print "Tree collapsed" and "Tree expanded" when the corresponding event occurs as you can see in the following source code:

```
tree.addTreeListener(new TreeListener() {  
    public void treeCollapsed(TreeEvent e) {  
        System.out.println("Tree collapsed.");  
    }  
  
    public void treeExpanded(TreeEvent e) {  
        System.out.println("Tree expanded.");  
    }  
});
```

Figure 8.41: TreeListener code sample

The resulting program should look something like this:

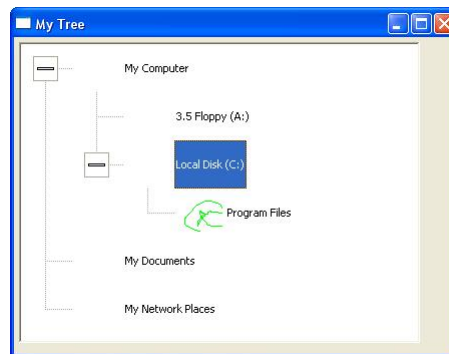


Figure 8.42: Tree resulting from the previous codes

(Note that if you used a tree with the `SWT.CHECK` style bit constant, the `TreeItems` `getChecked()` method could have been used to test whether a particular *Item* in the tree had been checked.)

8.3.4 TabFolder

The *TabFolder* widget is usually used to select a page in a set of pages. Once again, although it is a *Composite* you are not allowed to add *Composite* widgets to it, except if it is of type *TabItem*. There are the two important steps to create a *TabItem*:

1. if you want a *Label* on the tab, you need to use the `setText()` method
2. in order to populate the *TabItem*, you need to set the *Control* which it contains, by using the `setControl()` method

In the following example, `buttonComp` is a *Composite*. Since you can only set one *Control* into a *TabItem*, a *Composite* is usually desired. And then you can add several *Control* widgets to the *Composite* that will appear on the specified *TabItem*. In this code, you will see that *Events* for tab changing are built into the *TabItem* and *TabFolder* controls.

```
// create the TabFolder
TabFolder tabFolder1 = new TabFolder(shell, SWT.NONE);
tabFolder1.setBounds(10, 10, 270, 250);

// set up the button tab
Composite buttonComp = new Composite(tabFolder1, SWT.NONE);

Button button1 = new Button(buttonComp, SWT.PUSH);
button1.setSize(100, 100);
button1.setText("Hello");
button1.setLocation(0, 0);
Button button2 = new Button(buttonComp, SWT.ARROW);
button2.setBounds(150, 0, 50, 50);

TabItem item1 = new TabItem(tabFolder1, SWT.NONE);
item1.setText("Buttons");
item1.setControl(buttonComp);

// set up the label tab
Composite labelComp = new Composite(tabFolder1, SWT.NONE);

Label label1 = new Label(labelComp, SWT.NONE);
label1.setText("Here are some labels for your viewing pleasure");
label1.setBounds(0, 0, 250, 20);
Label label2 = new Label(labelComp, SWT.NONE);
label2.setText("A label is a fine fingered fiend");
label2.setBounds(0, 40, 200, 20);

TabItem item2 = new TabItem(tabFolder1, SWT.NONE);
item2.setText("Labels");
item2.setControl(labelComp);
```

Figure 8.43: TabFolder sample code

As said, this example shows how to create a *TabFolder* with two *TabItems* – one for buttons and one for labels. On the left, you see what it looks like when the window initially appears, on the right, you see what the screen looks like when you click on the "Labels" tab.

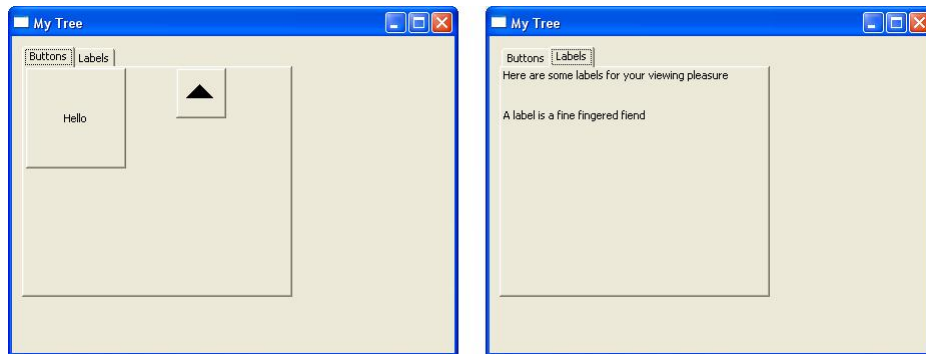


Figure 8.44: TabFolder resulting from the previous code

8.3.5 CoolBar

The *CoolBar* widget provides an area in which you add items that would be dynamically re-positionable. Obviously, you can add one or more *ToolBar* widgets to a *CoolBar*. The *CoolBar* can contain one or more *CoolItems*. And once again, although a *CoolBar* is a *Composite* widget, it is not allowed to add other composite classes because the sub-elements of a *CoolBar* must be of the type *CoolItem*. We can place whatever *Controls* we like on each *CoolItem*. In the following example, we will create in a standard way a bordered *CoolBar* and four *CoolItems* – one empty *CoolItem*, two *CoolItems* containing *Buttons* and a final *CoolItem* containing a *ToolBar* – specifying “bar” as the *CoolBar* to which they should be added.

```
final CoolBar bar = new CoolBar(shell, SWT.BORDER);
CoolItem item1 = new CoolItem(bar, SWT.NONE);
CoolItem item2 = new CoolItem(bar, SWT.NONE);
CoolItem item3 = new CoolItem(bar, SWT.NONE);
CoolItem item4 = new CoolItem(bar, SWT.NONE, 2);
```

Figure 8.45: CoolBar code sample

Then we have to go through the creating process of every *Control* we will place on the *CoolItems*. The first is a flat bordered button, that will be used to demonstrate how to lock and to unlock of the *CoolBar*.


```
Button button1 = new Button(bar, SWT.FLAT | SWT.BORDER);
button1.setText("Button");
button1.pack();
```

Figure 8.46: CoolBar Button source code example 1

When we use the `pack()` method on the *Button*, we set it to its correct size. The *Listener* that we next add to the button will call `bar.setLocked(boolean)` to lock (or unlock) the *CoolBar* each time it is clicked on. This means that when you click on it at first, it locks and the *ToolBar* removes its ability to click and drag the *CoolItems* around within the bar.

```
button1.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent e) {
        bar.setLocked(!bar.getLocked());
    }
});
```

Figure 8.47: CoolBar SelectionListener code sample

The next button is a simple *button* placed in its *CoolItem*.

```
Button button2 = new Button(bar, SWT.PUSH);
button2.setText("Another button");
button2.pack();
```

Figure 8.48: CoolBar Button source code example 2

The next piece of code places a basic 2-button *ToolBar* on its own *CoolItem*.

```
ToolBar tools = new ToolBar(bar, SWT.NONE);
ToolItem b1 = new ToolItem(tools, SWT.FLAT);
b1.setText("Tool");
ToolItem b2 = new ToolItem(tools, SWT.FLAT);
b2.setText("Bar");
tools.pack();
```

Figure 8.49: ToolBar code sample

Then, once we have created all the *Controls* we wanted, we can use: the `setControl()` method to associate the *Controls* with the right *CoolItems*, the `setSize()` method to set the initial size of each *Control* or the `setMinimumSize()` method called on `item3` to ensure that the *CoolItem* containing the *ToolBar* does not shrink beneath the size we request – in this case is the size of the *ToolBar* itself when the user rearranges the *CoolBar*.

```
Point size = button1.getSize();
item1.setControl(button1);
item1.setSize(item1.computeSize(size.x, size.y));
size = button2.getSize();
item2.setControl(button2);
item2.setSize(item2.computeSize(size.x, size.y));
size = tools.getSize();
item3.setControl(tools);
item3.setSize(item3.computeSize(size.x, size.y));
item3.setMinimumSize(size);
```

Figure 8.50: CoolBar Point code sample

In the end, we can use the `setWrapIndices()` method to wrap some *CoolItems* onto the next row at a particular index and the `setSize()` is called on *Bar* to specify its size on the *Shell*.

```
bar.setWrapIndices(new int[] { 3 });
bar.setSize(300, 120);
```

Figure 8.51: CoolBar setWrapIndices method code sample

Here is what the *Coolbar* looks like initially (on the left) and after some rearrangement, a locking of the *CoolBar* (on the right):

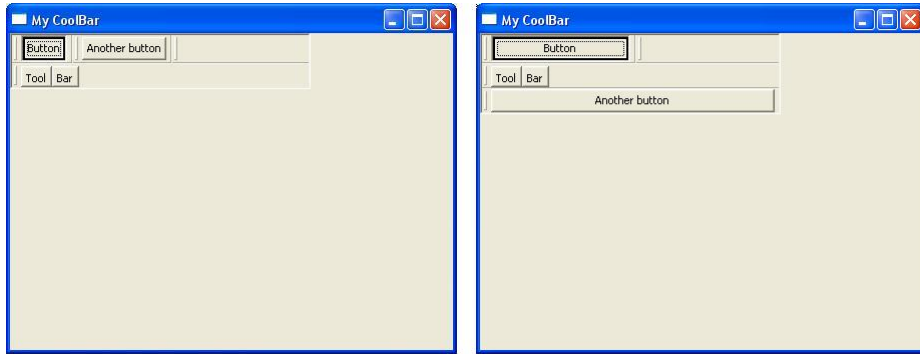


Figure 8.52: CoolBar resulting from the previous codes

(Note that the most common use of *CoolBars* is to create dynamically rearrangeable *Toolbars*. For example, they are use in Eclipse. This is how Eclipse and TeXnicCenter use them.)

8.4 Menu Widgets

8.4.1 Menu

In this section, we will look at two examples of menus. The first will be the most basic use, to create a simple menu with a selectable option on it, and capture the event generated when it is selected. The second – more complex than the first – will demonstrate some of the more complicated aspects of menus, including sub-menus, accelerators, mnemonics, additional Events that may be captured, and the various different types of menu items available.

But first, remark that the following style bit constants are available: `SWT.PUSH`, `SWT.CHECK` to get a checkbox item, `SWT.RADIO` to get a radio button item, `SWT.SEPARATOR` to provide an non-selectable dividing horizontal line between other items and `SWT.CASCADE`.

Simple Menu

When you want to create the menu bar that spans the top of your Shell use the following code which uses *Menu* and *MenuItem* widgets:

```
Menu menu = new Menu(shell, SWT.BAR);
shell.setMenuBar(menu);
```

Figure 8.53: Menu (simple) code source example 1

The style bit constant for the menu bar is `SWT.BAR`.

Then, let us add an *MenuItem* to this *Menu* bar. We will create a "File" *MenuItem*. (Note that if you launch your application now with only these lines, you will see a *Menu* bar at the *Shell* top with only the "File" option displayed. But if you select one of the options, nothing will happen. So, verify that the style bit constant is `SWT.CASCADE`. Otherwise you will be unable to attach a drop-down *Menu* to it.

```
MenuItem file = new MenuItem(menu, SWT.CASCADE);
file.setText("File");
```

Figure 8.54: MenuItem (simple) source code example 1

We will now create a *Menu* to the File option and attach *MenuItems* to it. (Note that the style constant for the menu must be `DROP_DOWN`)

```
Menu filemenu = new Menu(shell, SWT.DROP_DOWN);  
file.setMenu(filemenu);
```

Figure 8.55: Menu (simple) source code example 2

Finally, we can now add a *MenuItem* to our file *Menu*, and then associate an action with it to create a basic selectable *MenuItem* on the "File" *Menu*.

```
MenuItem actionItem = new MenuItem(filemenu, SWT.PUSH);  
actionItem.setText("Action");
```

Figure 8.56: MenuItem (simple) source code example 2

Then, in order to be notified when this option is selected, we add a listener that will output the words "Action performed!" to the console, whenever the "Action" option is chosen, like so:

```
actionItem.addListener(SWT.Selection, new Listener() {  
    public void handleEvent(Event e) {  
        System.out.println("Action performed!");  
    }  
});
```

Figure 8.57: Menu (simple) Listener code source example 1

Repeating these steps with more Menus and MenuItems is all you need to do to create a useful set of simple menus for your program.

And on the following page, you will see the resulting menu:

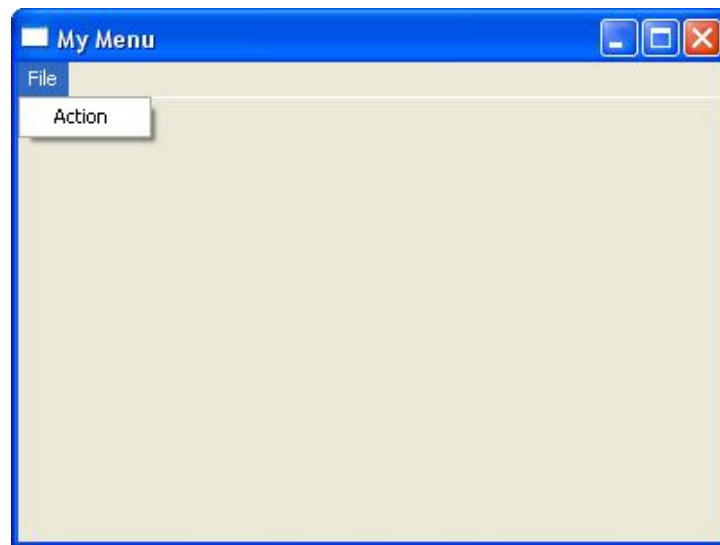


Figure 8.58: Menu (simple) resulting from the previous codes

Advanced Menu

Let us now start the longer example with the same initial code as the first example. The following code demonstrates how to use the `SWT.CHECK`, `SWT.SEPARATOR` and `SWT.RADIO` options. It will create a separator and a CheckBox-style *MenuItem* followed by radio-button styled *MenuItem*.

```
MenuItem separator = new MenuItem(filemenu, SWT.SEPARATOR);  
final MenuItem radioItem = new MenuItem(filemenu, SWT.RADIO);  
radioItem.setText("Radio");  
final MenuItem checkItem = new MenuItem(filemenu, SWT.CHECK);  
checkItem.setText("Check");
```

Figure 8.59: MenuItem (advanced) source code example 3

In the next few lines we add *Listeners* to the Check- and Radio- *MenuItems*, which output their current values (true or false) to the console.

```
radioItem.addListener(SWT.Selection, new Listener() {
    public void handleEvent(Event e) {
        System.out.println("Radio item toggled to:"
            + radioItem.getSelection());
    }
});
checkItem.addListener(SWT.Selection, new Listener() {
    public void handleEvent(Event e) {
        System.out.println("Check item toggled to:"
            + checkItem.getSelection());
    }
});
```

Figure 8.60: Menu (advanced) Listener code source example 2

Let us now add a sub-menu that branches off of the file menu, which is very similar to how we attached our "File" *Menu* to the *Menu* bar. Then we create a new *MenuItem* with the `SWT.CASCADE` style bit constant, and a new *Menu* to attach to it. We then call `setMenu()` method on the *MenuItem* to which we attach the new menu.

```
MenuItem cascadeItem = new MenuItem(filemenu, SWT.CASCADE);
cascadeItem.setText("Cascade");
Menu submenu = new Menu(shell, SWT.DROP_DOWN);
cascadeItem.setMenu(submenu);
```

Figure 8.61: MenuItem (advanced) code source example 4

In the next few lines we add a *MenuItem* to our new sub-menu. This new *MenuItem* that we have added will be called "SubAction".

Notice that in the `setText()` method call, there is an ampersand(&) before the letter 'S'. This will create a mnemonic for the SubAction command, so that when the submenu is displayed, if you press the *S_KEY* it will select it as if you had used the mouse. Then the call to `setAccelerator()` associates the SubAction *MenuItem* with the key combination *Control-S*, so the user can execute the SubAction command without bringing up the menu.

```
final MenuItem subactionItem = new MenuItem(submenu, SWT.PUSH);
subactionItem.setText("&SubAction\tCtrl+S");
subactionItem.setAccelerator(SWT.CTRL + 'S');
```

Figure 8.62: MenuItem (advanced) code source example 5

Then we add an additional option to the sub-menu which will be used to demonstrate the enabling and disabling operations of the *MenuItems*. This check box will enable/disable the SubAction we just added.

```
final MenuItem enableItem = new MenuItem(submenu, SWT.CHECK);
enableItem.setText("Enable SubAction");
```

Figure 8.63: MenuItem (advanced) code source example 6

There exist several ways to add *Listeners* that we did not look at in our simple example. One of these is the ability to add listeners to the *Menus* (rather than to the *MenuItems* as we did before.) Let us add a *MenuListener* to our sub-menu, which notifies us whenever it is shown or hidden.

```
submenu.addMenuListener(new MenuListener() {
    public void menuShown(MenuEvent e) {
        System.out.println("SubMenu shown");
    }

    public void menuHidden(MenuEvent e) {
        System.out.println("SubMenu hidden");
    }
});
```

Figure 8.64: Menu (advanced) Listener code source example 3

Then let us demonstrate how it is possible to enable and disable *MenuItems* by using another check box *MenuItem* where we disable SubAction to begin, so its text is shown shaded in the menu. After we have disabled it you cannot select the option, and its accelerator also no longer works. The rest of the code sets up the listener that will disable or enable the SubAction *MenuItem* depending on whether the "Enable SubAction" check box is checked.


```
subactionItem.setEnabled(false);
enableItem.addListener(SWT.Selection, new Listener() {
    public void handleEvent(Event e) {
        System.out.println("Toggling \"Enable SubAction\" to "
            + enableItem.getSelection());
        subactionItem.setEnabled(enableItem.getSelection());
    }
});
```

Figure 8.65: Menu (advanced) sub-action code source example 1

Remark that in the previous examples, we used a generic *Listener*. Let us now use the more specific *SelectionListener* (seen in section 8.1.1). And as before we will print a message to the console at each activation.

```
subactionItem.addSelectionListener(new SelectionListener() {
    public void widgetSelected(SelectionEvent e) {
        System.out.println("SubAction performed!");
    }

    public void widgetDefaultSelected(SelectionEvent e) {
    }
});
```

Figure 8.66: Menu (advanced) sub-action code source example 2

Another interesting *Listener* is the *ArmListener*, that is fired when a *MenuItem* is highlighted with the mouse or keyboard, but not yet selected. The following code prints a message when the SubAction is armed.

```
subactionItem.addArmListener(new ArmListener() {
    public void widgetArmed(ArmEvent e) {
        System.out.println("SubAction armed!");
    }
});
```

Figure 8.67: Menu (advanced) sub-action code source example 3

The final listener type is the *HelpListener*. This mean that by pressing the *F1/HELP_KEY*, the *HelpListener* is activated.

```
subactionItem.addHelpListener(new HelpListener() {  
    public void helpRequested(HelpEvent e) {  
        System.out.println("Help requested on SubAction");  
    }  
});
```

Figure 8.68: Menu (advanced) sub-action code source example 4

You can see just below the result of the preceding codes. If you select the "Enable" SubAction option, you will enable SubAction and if you type *Ctrl+S*, this will execute SubAction as well, but only if Enable SubAction is checked. Note that all the listeners print messages to the console when they are activated.

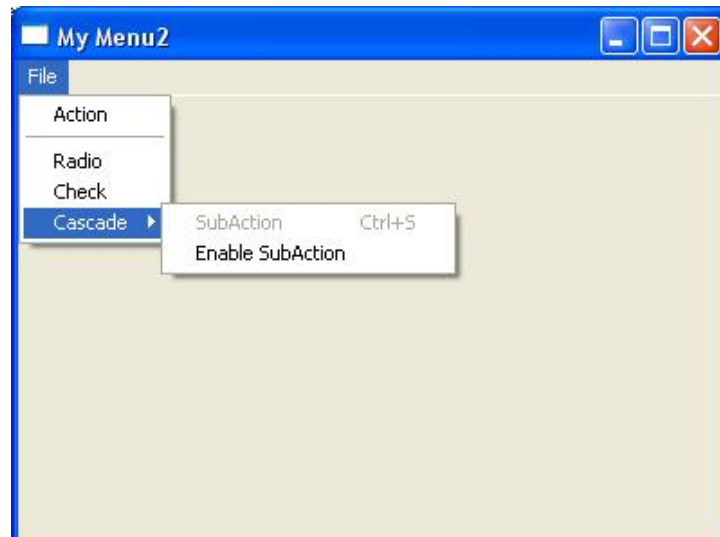


Figure 8.69: Menu (advanced) resulting from the previous codes

It exists other facilities available for Menus and MenuItems, but we have attempted to cover the majority of the useful ones. Next we will look briefly at a simple pop-up menu.

8.4.2 Pop-Up Menu

The Pop-up *Menus* are usually used for the context-sensitive menus on different areas of the screen, rather than making the user go to a *Menu* bar at the top of the screen and hunt for the option he wants. The right-click on

the *Composite* that has a pop-up menu associated is a kind of shortcut that will make appear a floating menu beside your mouse allowing you to select an option from a list like a regular menu. In the example of this section, a *Button* will be created in a *Composite* on a *Shell*. Then two Pop-up Menus will be created. The first will be associated with the *Shell* and the *Composite* while the other with the *Button*.

So, we create a *Menu*, as we did in the section 8.4.1, except that its style bit constant will be `SWT.POP_UP`. We then add an *MenuItem* and attach a *Listener* to capture its selection events.

```
Menu popupmenu = new Menu(shell, SWT.POP_UP);
MenuItem actionItem = new MenuItem(popupmenu, SWT.PUSH);
actionItem.setText("Other Action");
actionItem.addListener(SWT.Selection, new Listener() {
    public void handleEvent(Event e) {
        System.out.println("Other Action performed!");
    }
});
```

Figure 8.70: PopUp menu source code example 1

Then we repeat the process, with the *Menu* associated with the *Button*.

```
Menu popupmenu2 = new Menu(shell, SWT.POP_UP);
MenuItem buttonItem = new MenuItem(popupmenu2, SWT.PUSH);
buttonItem.setText("Button Action");
buttonItem.addListener(SWT.Selection, new Listener() {
    public void handleEvent(Event e) {
        System.out.println("Button Action performed!");
    }
});
```

Figure 8.71: PopUp menu source code example 2

Next we create the *Composite* and *Button*, and place them on the *Shell*.

```
Composite c1 = new Composite(shell, SWT.BORDER);  
c1.setSize(100, 100);  
c1.setLocation(25, 25);  
Button b = new Button(c1, SWT.PUSH);  
b.setText("Button");  
b.setSize(50, 50);  
b.setLocation(25, 25);
```

Figure 8.72: PopUp menu Composite code sample

And finally, let us set the Pop-up menus.

```
b.setMenu(popupmenu2);  
c1.setMenu(popupmenu);  
shell.setMenu(popupmenu);
```

Figure 8.73: PopUp menu setMenu()

Now if we run the code, and right-click on the Button, it will display one menu, but if we right-click the Composite or Shell, we get another menu with different options. In the following screen shots we can see the Pop-up menu that is displayed when right-clicking on the Button (on the left) and outside of it (on the right).

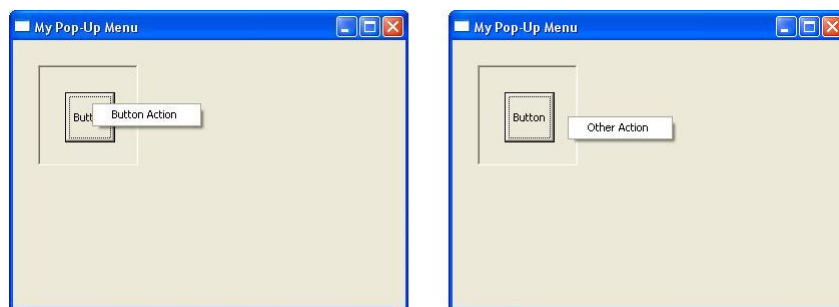


Figure 8.74: PopUp menu resulting from the previous codes

Although this is a simple example, Pop-up menus can be just as complex as the *Menus* in the section 8.4.1

8.5 A summary of controls having items

To summarize the preceding chapter, there exist *Controls* that only accept sub components of type *Item*. These *Controls* are listed below.

Widget	Item	Description
CoolBar	CoolItem	Items are selectable, dynamically positionable areas of a CoolBar
Menu	MenuItem	Items are selections under a menu
TabFolder	TabItem	Items are Tabs in a TabFolder
Table	TableItem TableColumn	Items are Rows in a table
ToolBar	ToolItem	Items are Buttons on the tool bar
Tree	TreeItem	Items are nodes in a tree

Figure 8.75: Widget needing Item summary

Chapter 9

Layouts

What is and what does a layout? - It automatically controls the position and the size of all the widgets inside of a *Composite*. In SWT, the size of a widget inside a composite is not automatically set (the layout can do this for you). Each composite requires a layout to show its children controls that need to be set or SWT will not be able to set the *Controls* size and position inside a composite and our controls will not be visible.

The layout Manager classes are inherited from an abstract class: *Layout*. Some of them allow us to set different properties for each control. Those *Layout* classes have an addition object that can be set separately for each *Control* widget inside of a *Composite*. The *Control* class provides a standard method, `setLayoutData(Object obj)`, to set this addition parameter. You have to set an appropriate layout data object for each control, otherwise it throws a classcast exception.

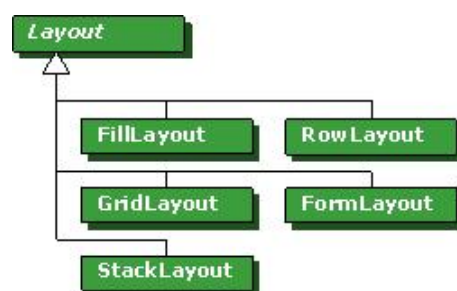


Figure 9.1: SWT Layout manager hierarchy

All the component in the *Composite* are affected by the layout properties. Some layouts support layout data objects to set different properties for each *Control* within a *Composite*. These properties can be set by using a layout's

member variables. If you are familiar with SWING, you would probably search for methods to set and get these member variables. In SWT, it is different: you read and write to those variables directly.

9.1 FillLayout

The *FillLayout* is the simplest (and set by default) *Layout*. If there is only one component, the *FillLayout* makes it fill the whole available area of the composite. If there are more than one components, it forces them all to get the same size in a column (or a line, depending on the style bit constant of the *Layout*). No options are available to control the spacing, the margins or the wrapping.

The style bit constants for this layout are: `SWT.HORIZONTAL` (used by default) or `SWT.VERTICAL` to position the controls either in a single row or a column. You can set this in the constructor, or later as a field of the layout.

```
FillLayout fillLayout = new FillLayout();
fillLayout.type = SWT.VERTICAL;

// or you can declare it by this way
FillLayout fillLayout2 = new FillLayout(SWT.VERTICAL);
```

Figure 9.2: FillLayout code sample

The *FillLayout* is usually used for *ToolMenu* bar, *Group* or *Composite* having only one child. Besides that, it might be used to stack the check boxes or radio buttons in a group.

The width and height of a *Control* within a *Composite* are determined by the parent composite. Initially, the height of any control is equal to the highest, and the width of a control is equal to the widest.

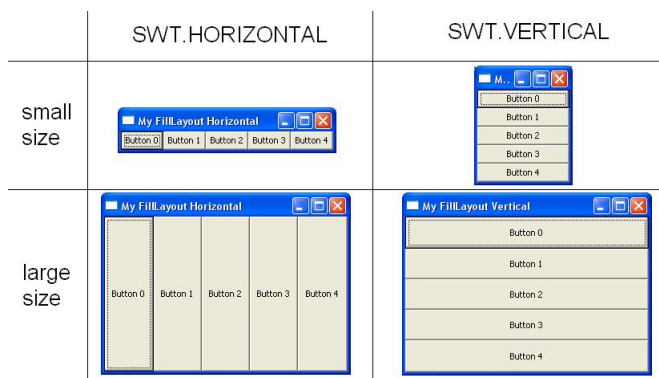


Figure 9.3: FillLayout resulting from window resizing

9.2 RowLayout

The *RowLayout* is very similar to *FillLayout* (seen in the chapter 9.1). It positions the *Controls* in the same way as the *FillLayout*: in rows or columns. You can use the following code to use a *RowLayout* in your application:

```
RowLayout rowLayout = new RowLayout();  
shell.setLayout(rowLayout);
```

Figure 9.4: RowLayout code sample

And in addition to the *FillLayout*, the *RowLayout* provides configuration fields to control the position of a control within a composite as:

- if the number of widgets does not fit in a row, it wraps them (*wrap* field set to true by default) as you can see it (from left to right)



Figure 9.5: RowLayout wrapping sample

- configurable margins (*marginBottom*, *marginRight*, *marginTop* and *marginLeft* fields are set to 3 by default)

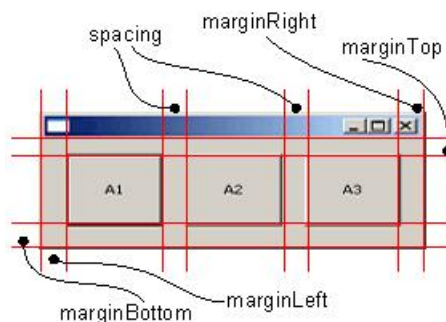


Figure 9.6: RowLayout margins and spacing sample

- a *RowData* is provided

```
Button[] buttons = new Button[5];
for (int i = 0; i < 5; i++) {
    buttons[i] = new Button(shell, SWT.PUSH);
    buttons[i].setText(String.valueOf(i));
    RowData rowData = new RowData(i*20, i*30);
    buttons[i].setLayoutData(rowData);
}
```

Figure 9.7: RowLayoutData code sample

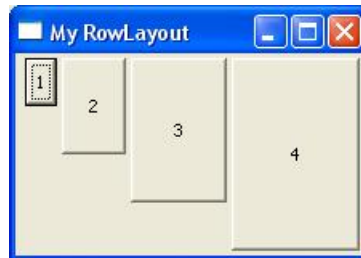


Figure 9.8: RowLayoutData sample

Note that you can also use the following *RowLayout* fields: **pack** to force all the component of a *Composite* to have the same size and **justify** to spread the widgets of a *Composite* across the available space.

9.3 GridLayout

The *GridLayout* is the most useful, flexible and powerful layout. It lays out the widgets in grids and has a lot of configurable properties we will see below. To set a different property for a specific cell, you need to create a *GridData* object and set this as the control's layout data object.

To use this *layout* in your application, you can use the following code sample:

```
GridLayout gridLayout = new GridLayout();
shell.setLayout(gridLayout);
gridLayout.numColumns = 5;
```

Figure 9.9: GridLayout code sample

The *GridLayout* provides configuration fields to control the position of a control within a composite as:

- The `numColumns` field must be set to specify the number of columns in the *GridLayout*. If the number of components added is bigger than the components number of the composite, the layout adds them to a new row. E.g. five components added to a GridLayout that only has four columns will make that the fifth component will be added to the second row. The `marginWidth` and `marginHeight` to set the margins values, `horizontalSpacing` and `verticalSpacing` to set the spacings, can be set, too.

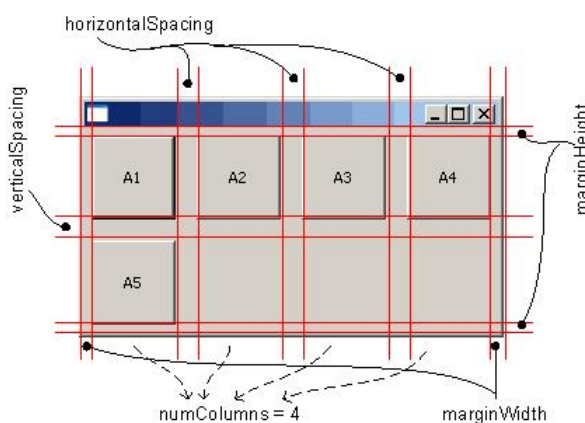


Figure 9.10: GridLayout number of columns and spacings sample

- The `makeColumnEqualWidth` property set to "true" makes the columns the same size. (If you want the widgets also to be the same size, you should need to use *GridData* object's horizontal/vertical alignment properties). You can see below a window without this property (on the left), and after we activate this last one (on the right).

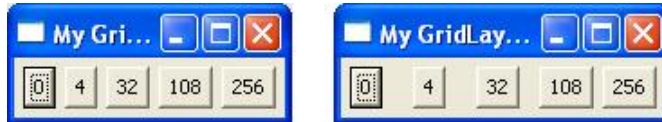


Figure 9.11: GridLayout with column of equal width sample

- A *GridData* which set a different property for each widget, is provided by the *GridLayout*. The most useful *GridData* properties are shown below:

```
Button[] buttons = new Button[5];
for (int i = 0; i < 5; i++) {
    buttons[i] = new Button(shell, SWT.PUSH);
    buttons[i].setText(String.valueOf(i * i * i * 4));
    GridData gridData = new GridData();
    buttons[i].setLayoutData(gridData);
}
```

Figure 9.12: GridData code sample

- The `horizontalSpan` (or `verticalSpan`) property sets how many column(s) (or lines) we want our component takes. You can see the effect on the first component when this property is set to 1 (by default, on the left) or to 2 (on the right).

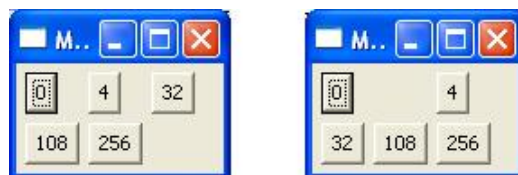


Figure 9.13: GridData horizontal spanning sample

- The `grabExcessHorizontalSpace` (or `grabExcessVerticalSpace`) property, when set to true, makes the width of the grids will be as large as possible. You can see this effect on the first component of the bottom picture, instead of the top picture for which this property is set to false.



Figure 9.14: GridData with the largest cell width sample

- The `horizontalAlignment` (or `verticalAlignment`) property sets the alignment of the *Control*. You can use this in combination with the `grabExcessHorizontalSpace` property and see below what happens when it is set to "left" (top picture), "center" (centered picture) or "right" (bottom picture).



Figure 9.15: GridData with horizontal alignment sample

9.4 FormLayout

The *FormLayout* is, as the *GridLayout*, a very flexible layout, but it works in a completely different way. Indeed, in contrast to *GridLayout*, *FormLayout* is independent from the complete layout. The position and size of the components depend on one *Control*. The properties you can set are `marginWidth` and `marginHeight` that allow to set the margin values. You can use the following code to manage your application with a *FormLayout*:

```
FormLayout formLayout = new FormLayout();
shell.setLayout(formLayout);
```

Figure 9.16: FormLayout code sample

Each widget within a composite having a *FormLayout* can have a *FormData* to set the different layout properties for a widget. The *FormData* object has the following properties: `width`, `height`, `top`, `bottom`, `left` and `right` whose functions seem to be obvious.

A *FormData* object can have 0 or 4 *FormAttachment* objects. The *FormAttachment* object can have the following properties: `alignment` to specify the alignment of the *Control* side that is attached to a *Control*, `control` to specify the *Control* to which the *Control* side is attached, `denominator` to specify the denominator (set by default to 100), `numerator` to specify the numerator and `offset` to specify the offset, in pixels, of the *Control* side from the attachment position.

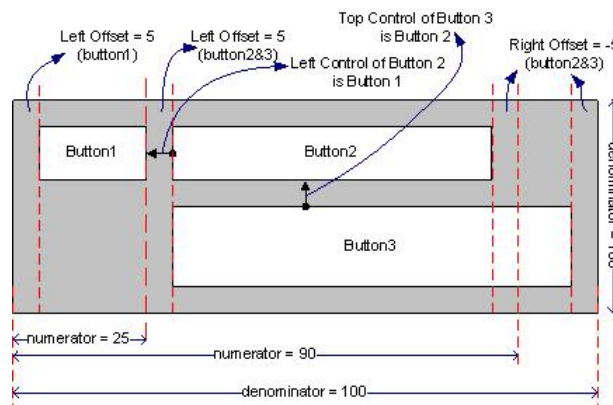


Figure 9.17: FormData and FormAttachement sample

The following code shows how to use the *FormLayout* manager:

```
Button button1 = new Button(shell, SWT.PUSH);
Button button2 = new Button(shell, SWT.PUSH);
Button button3 = new Button(shell, SWT.PUSH);
button1.setText("1");
button2.setText("2");
button3.setText("3");

FormData formData1 = new FormData();
formData1.left = new FormAttachment(0, 5);
formData1.right = new FormAttachment(25, 0);
button1.setLayoutData(formData1);

FormData formData2 = new FormData();
formData2.left = new FormAttachment(button1, 5);
formData2.right = new FormAttachment(90, -5);
button2.setLayoutData(formData2);

FormData formData3 = new FormData();
formData3.top = new FormAttachment(button2, 5);
formData3.bottom = new FormAttachment(100, -5);
formData3.left = new FormAttachment(25, 5);
formData3.right = new FormAttachment(100, -5);
button3.setLayoutData(formData3);
```

Figure 9.18: FormData code sample

You can see the result of the previous code on the following page (on the right, when you try to increase the size of the window)

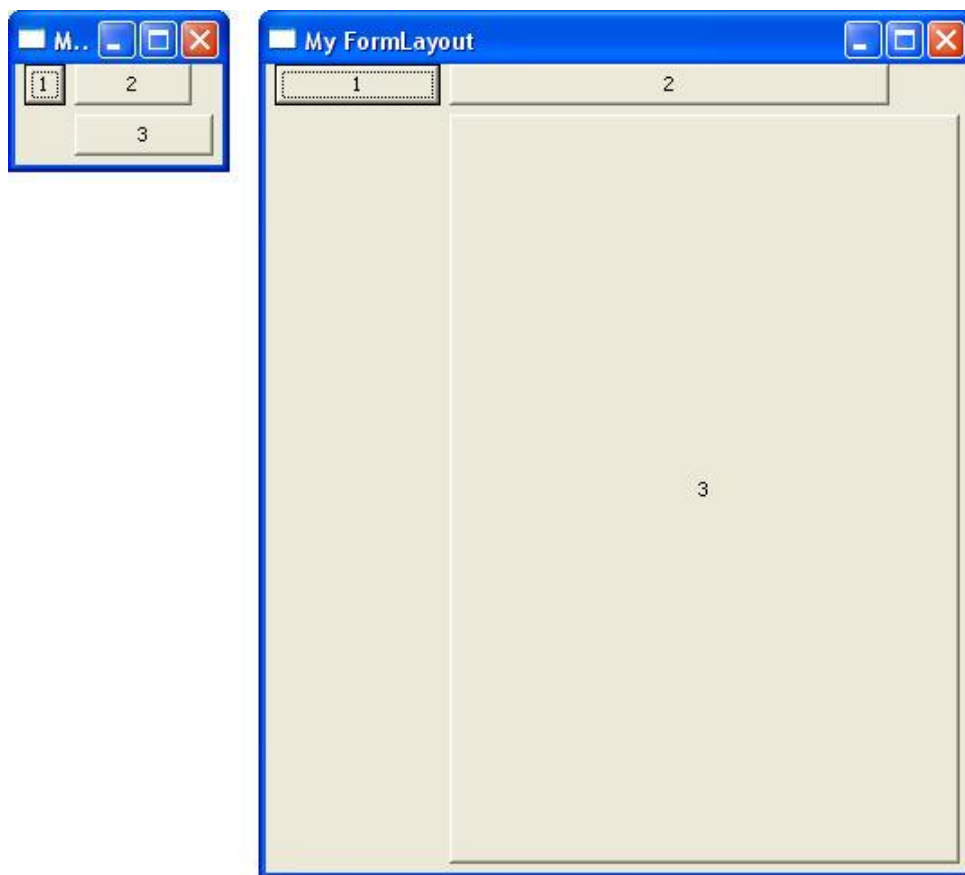


Figure 9.19: FormData resulting from the previous code

9.5 StackLayout

The *StackLayout* is quite different from the other layout managers. Only one *Control* is displayed at a time instead of every *Controls* for the other layouts. We usually use the *StackLayout* for the property pages, wizards,... Its most usual properties are: `marginHeight`, `marginWidth` to set the margin values and `topControl` to specify the *Control* that is displayed at the top of the stack.

Here is an example to show you how to use a StackLayout:

```
shell.setLayout(new GridLayout());

final Composite parent = new Composite(shell, SWT.NONE);
parent.setLayoutData(new GridData());

final StackLayout stackLayout = new StackLayout();
parent.setLayout(stackLayout);

final Group[] group = new Group[3]; // 1
for (int i = 0; i < group.length; i++) {
    group[i] = new Group(parent, SWT.NONE);
    group[i].setText("Group " + (i + 1));

    GridLayout gridLayout = new GridLayout();
    gridLayout.numColumns = 4;
    group[i].setLayout(gridLayout);

    Character character = new Character((char) ('A' + i));
    for (int j = 10; j < 20; j++) {
        Button bArray = new Button(group[i], SWT.PUSH);
        bArray.setText(character + "." + j);
    }
}
stackLayout.topControl = group[0]; // 2
```

Figure 9.20: StackLayout code example 1/2

CHAPTER 9. LAYOUTS

```
stackLayout.topControl = group[0]; // 2

Button b = new Button(shell, SWT.PUSH);
b.setText("Show next group");
final int[] index = new int[1];
b.addListener(SWT.Selection, new Listener() {
    public void handleEvent(Event e) {
        index[0] = (index[0] + 1) % 3;
        stackLayout.topControl = group[index[0]]; // 3
        parent.layout(); // 4
    }
});
```

Figure 9.21: StackLayout code example 2/2

We have here three (1)*Groups* and we add *Button* controls to those groups. Then, we need to set the (2)initial `topControl`. We create an action *Listener* so that when the button is clicked it (3)changes the `topControl` and calls the `layout()` method of the (4)parent *Control*. This step is very important because if you do not call the `layout()` method, you will not be able to see any change.

And here is the result: from the left to the right, we only clicked on the "Show next group" button:

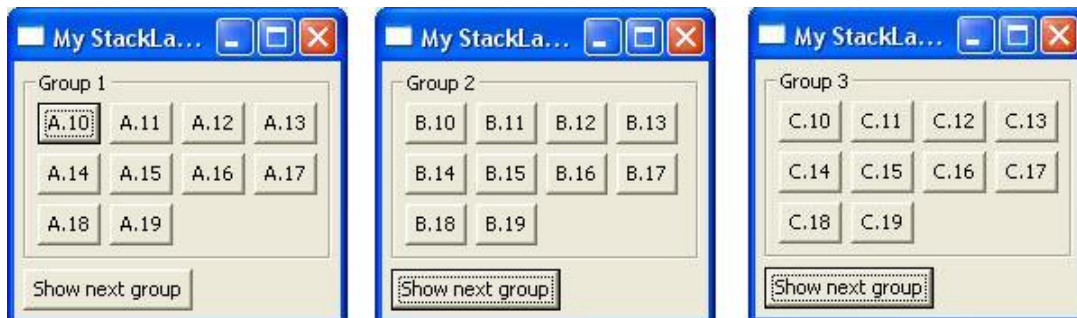


Figure 9.22: StackLayout resulting from the previous code

Appendix A

SWT editors

As you can see, during this tutorial, we decided to explain the SWT installation, philosophy and how to implement an application with this Java library developed by IBM. But, as visual editor for Swing and AWT are available, free SWT visual editors appeared at the end of the year 2004.

That is why we decided to make this appendix to give you some url address where to find free- or shareware versions. Most of the freeware versions are developed as Eclipse plug-ins because it is a library developed by IBM and specially used by the Eclipse environment (developed by IBM, too). But some other – shareware this time – can be used by other IDE as WebSphere.

- <http://dev.eclipse.org/viewcvs/indextools.cgi/vcp-home/WebContent/docs/newAndNoteworthy/1.0.0-final/vcp-news-final.html?rev=1.2> (freeware)
- <http://www-106.ibm.com/developerworks/opensource/library/os-ecca/> (freeware)
- http://www.instantiations.com/swt-designer/home_content.html (shareware)

Appendix B

An interesting custom widget example

To display a picture on a screen is quite easy, but sometimes, we would like to be allowed to zoom in (or zoom out of) it with a *scale* widget and obviously, that scrollbar appears when the picture will be too large for the *Canvas* widget. The example developed in this chapter will do this: a zoomed scrolled viewer widget. This widget will contain a *Canvas*, *ScrollBars*, and a *scale* grouped together in one *Composite*.

Notice that a *ScrolledComposite* has already been created. In this example, we will use this *ScrolledComposite* custom *Control* within our own custom *Control*, instead of building it from scratch.

So, let us have a look on the following page at the skeleton we will use. For this, we decided to create every widget in the constructor using a *GridLayout* (shown in section 9.3). To initialize and place our widgets, we will do this by calling the `setup()` method that will act on the *Composite*, the *Scale* and the *Label*.

```

public class ZoomedScrolledCanvas extends Composite {
    Label zoom;
    ScrolledComposite comp;
    Canvas canvas;
    Scale scale;
    Image image;
    int width, height;

    public ZoomedScrolledCanvas(Composite parent, int style) {
        super(parent, style);
        comp = new ScrolledComposite(this, SWT.BORDER | SWT.H_SCROLL
            | SWT.V_SCROLL);
        canvas = new Canvas(comp, 0);
        scale = new Scale(this, 0);
        zoom = new Label(this, 0);
        setLayout(new GridLayout());

        // Do not forget to put these 3 method calls at this place in your code
        // setupComp();
        // setupScale();
        // setupZoom();
    }
}

```

Figure B.1: custom ZoomedScrolledCanvas code sample

- To setup the composite, we must choose a layout data and attach to it the canvas so that we can draw the image within the scrollbars.

```

public void setupComp() {
    comp.setAlwaysShowScrollBars(true);
    comp.setLayoutData(new GridData(GridData.FILL_BOTH));
    comp.setContent(canvas);
    layout(true);
}

```

Figure B.2: custom setComp() method code sample

- Then, to setup the scale, we need to specify the maximum, minimum, increment, and current selection (and the layout data).


```
public void setupScale() {  
    scale.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));  
    scale.setMinimum(0);  
    scale.setMaximum(100);  
    scale.setIncrement(10);  
    scale.setSelection(50);  
    layout(true);  
}
```

Figure B.3: custom setupScale() method code sample

- And finally, to setup the label, we need to give its layout data and some text.

```
public void setupZoom() {  
    zoom.setText("Zoom Factor = 0");  
    zoom.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));  
    layout(true);  
}
```

Figure B.4: custom setupZoom() method code sample

The next thing we must do is to draw the image onto the canvas. For this, we will first set the size of the *Canvas* and of the *ScrolledComposite* and an image that will appear in its full size. We then need to add a paint listener to (re-)draw the image that will be scaled to the size of the *Canvas*. This is really useful, because when we are zooming on the picture, we will only need to change the size of the canvas and when this happens, a paint event will be triggered and the image will be redrawn every time using the size of the *Canvas*. Note that we also have a dispose listener to dispose of the image when the *Canvas* is disposed.

```
public void setImage(String imageString) {
    image = new Image(this.getDisplay(), imageString);
    canvas.setSize(comp.getSize().x, comp.getSize().y);
    canvas.setLocation(0, 0);

    canvas.addPaintListener(new PaintListener() {
        public void paintControl(PaintEvent e) {
            GC gc = e.gc;
            width = comp.getSize().x;
            height = comp.getSize().y;
            gc.drawImage(image, 0, 0, image.getBounds().width, image
                .getBounds().height, 0, 0, canvas.getSize().x, canvas
                .getSize().y);
        }
    });

    canvas.addDisposeListener(new DisposeListener() {
        public void widgetDisposed(DisposeEvent e) {
            image.dispose();
        }
    });

    // here, you will put the next piece of code

    redraw();
}
```

Figure B.5: custom setImage() method code sample

Now that our appearance *Control* is set up, we will add a *SelectionListener* to the *Scale* to zoom in or out of the image depending on the changed value on the *Scale*. In this *Listener*, we calculate a new size based on the zoom factor and resize the canvas. (When the canvas is resized in the *SelectionListener*, a *PaintEvent* will be generated by the *PaintListener* placed on our *Canvas*, and so the image will be drawn at its new size.

APPENDIX B. AN INTERESTING CUSTOM WIDGET EXAMPLE

```
scale.addSelectionListener(new SelectionAdapter() {  
    public void widgetSelected(SelectionEvent arg0) {  
        zoom.setText("Zoom Factor = " + (scale.getSelection() - 50));  
        int newWidth = width * (scale.getSelection()) / 50;  
        int newHeight = height * (scale.getSelection()) / 50;  
        canvas.setSize(newWidth, newHeight);  
    }  
});
```

Figure B.6: custom SelectionListener code sample

And finally, to see the widget in action, we use it in exactly the same way as we would use any widget.

```
Display display = new Display();  
Shell shell = new Shell(display);  
shell.setSize(400, 400);  
  
ZoomedScrolledCanvas zsc = new ZoomedScrolledCanvas(shell, SWT.NONE);  
zsc.setSize(300, 300);  
zsc.setLocation(10, 10);  
zsc.setImage("moi.JPG");  
  
shell.open();  
  
while (!shell.isDisposed()) {  
    if (!display.readAndDispatch())  
        display.sleep();  
}  
display.dispose();
```

Figure B.7: custom last step code sample

We set the size and the position of our widget just as we would with a basic widget. The only custom parameter you must set is the image. We set the image to be sunset.JPG. The following screen is generated from this code:

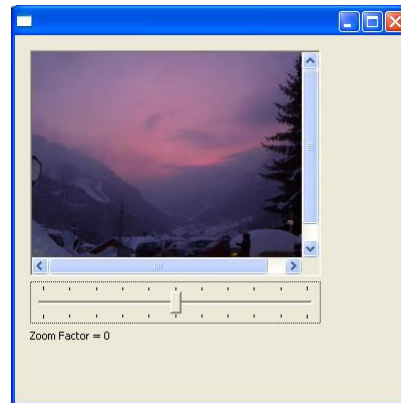


Figure B.8: custom application resulting from the previous codes

When we move the zoom scale down, the image gets smaller:

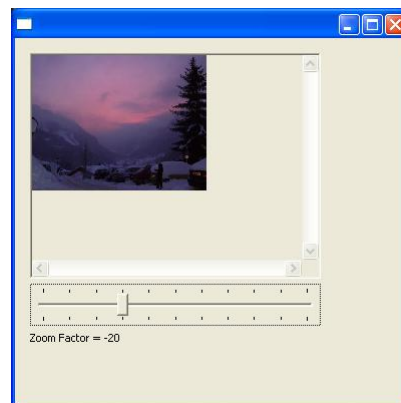


Figure B.9: custom application used 1

APPENDIX B. AN INTERESTING CUSTOM WIDGET EXAMPLE

When we move the zoom scale up, the image gets larger and we can scroll through it:

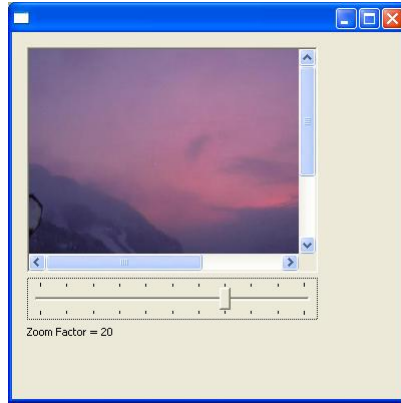


Figure B.10: custom application used 2

Bibliography

Eclipse Building Commercial-Quality Plug-ins (Clayberg, Rubel)
<http://www.eclipse.org>
<http://www.cs.umanitoba.ca/~eclipse/9-Custom.pdf>
<http://fr.wikipedia.org/wiki/SWT>
<http://java.sun.com/docs/books/tutorial/uiswing>
http://www.3plus4software.de/sma/Ein_SWT_Tutorial.html
<http://swt-tutorial.ch.vu>